Titre: Title:	Outils d'analyse de performance de systèmes d'exploitation avec partitionnement spatial et temporel
Auteur: Author:	Guillaume Champagne
Date:	2019
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Champagne, G. (2019). Outils d'analyse de performance de systèmes d'exploitation avec partitionnement spatial et temporel [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/4014/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/4014/
Directeurs de recherche: Advisors:	Michel Dagenais
Programme: Program:	Génie informatique

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Outils d'analyse de performance de systèmes d'exploitation avec partitionnement spatial et temporel

GUILLAUME CHAMPAGNE
Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie informatique

Août 2019

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Outils d'analyse de performance de systèmes d'exploitation avec partitionnement spatial et temporel

présenté par **Guillaume CHAMPAGNE**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

Gabriela NICOLESCU, présidente
Michel DAGENAIS, membre et directeur de recherche
Giovanni BELTRAME, membre

REMERCIEMENTS

Tout d'abord, je tiens à remercier mon directeur de recherche Michel Dagenais pour les nombreux conseils qu'il m'a offerts durant la réalisation de ce travail. Sa disponibilité, ses rétroactions rapides et son aide ont été essentielles afin de mener à bien ce projet.

Je tiens ensuite à remercier l'équipe de Mannarino Systems & Software Inc. pour leur grande disponibilité à répondre à toutes mes questions, le partage de leur grande expertise des systèmes temps réel en avionique, qui a permis d'orienter la recherche, ainsi que leurs conseils et commentaires tout au long du projet.

Je souhaite également remercier Mannarino Systems & Software Inc., le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG) ainsi que le Consortium de recherche et d'innovation en aérospatiale au Québec (CRIAQ) pour le soutien financier offert à mon projet.

Finalement, je tiens à remercier mes collègues du DORSAL pour tous les bons moments et les nombreuses pauses-café partagés.

RÉSUMÉ

Dans le domaine de l'avionique commerciale, les systèmes informatisés sont de plus en plus nombreux et complexes. Le partage des ressources devient alors une nécessité afin de limiter leur croissance en masse et en coût. Puisque ces systèmes embarqués évoluent dans un environnement imposant des contraintes temporelles et de sécurité très strictes, les systèmes d'exploitation temps réel utilisés doivent être conçus spécialement pour ces conditions. Dans ce contexte, le standard ARINC 653 définissant le comportement et les interfaces de systèmes avec partitionnement spatial et temporel a été publié. Ces systèmes d'exploitation fournissent les garanties nécessaires au partage des ressources dans un domaine temps réel critique.

Lors du développement d'applications sur ce genre de systèmes, les techniques de débogage classiques ne sont pas adaptées pour tous les types de problèmes pouvant survenir. En effet, l'arrêt complet de l'exécution d'un programme temps réel influence trop son comportement pour que la chaîne d'évènements menant au problème recherché soit observable. Par conséquent, l'enregistrement d'informations durant l'exécution pour une analyse a posteriori est une méthode avantageuse pour ce type de situation. Le traçage est une technique qui permet aussi de récupérer l'information sur le temps d'exécution de certaines composantes, ce qui peut être utile pour la caractérisation de performances du système. Pourtant, la disponibilité de tels outils pour les systèmes d'exploitation temps réel avec partitionnement spatial et temporel est très limitée en ce moment.

Dans ce mémoire, nous présentons un environnement d'analyse de traces pour les systèmes temps réel avec partitionnement spatial et temporel. Nous considérons l'exécution de processus à l'intérieur de partitions et l'ordonnancement de ces deux éléments par le système d'exploitation. Grâce à l'information contenue dans une trace, un historique des états des processus à travers le temps est reconstruit. La solution est testée sur Linux et Xen, deux systèmes pour lesquels nous créons des configurations s'approchant du comportement d'un système d'exploitation avec partitionnement spatial et temporel.

Pour montrer l'utilité de cet outil de visualisation pour le diagnostic de problèmes de performances, nous développons un banc d'essai adapté aux systèmes avec partitionnement spatial et temporel. Le banc d'essai est conçu pour être utilisé de façon indépendante ou de pair avec l'environnement d'analyse de traces. Cette deuxième option permet d'identifier automatiquement les valeurs extrêmes dans les résultats afin d'observer l'état du système lors de leurs occurrences. Avec cette fonctionnalité, nous montrons comment notre solution permet aussi le diagnostic de problèmes causés par l'exécution concurrente d'applications mal isolées.

ABSTRACT

Computer systems in commercial avionics keep increasing in number and complexity. To reduce their cost and mass, it becomes highly interesting to share the resources between the different subsystems. Since the applications composing these subsystems must meet particularly stringent time and safety constraints, the operating systems managing them must be conceived with these specificities in mind. The ARINC 653 standard was published to regulate the development of such operating systems that provide very strong guarantees about the space and time isolation of the running applications, to enable the sharing of resources between them.

Traditional debugging techniques are not efficient for every type of problems that developers may encounter while designing real-time applications. Entirely stopping the execution of the studied application is not always an option, since it may impact the chain of events leading to the problematic situation. An alternative to this is to record information about the execution, while the applications are running, and to analyze it offline. This information is also useful to compute performance metrics about the system. Unfortunately, the availability of such tools is very limited for operating systems with space and time partitioning.

We present a trace analysis framework for operating systems with space and time partitioning. The visualization tool developed covers the scheduling of processes, and partitions containing theses processes. With the information available in the operating system trace, a state history of the processes is built and displayed to the user. The solution is tested on Linux and Xen, two systems for which we created configurations to approximate as accurately as possible the behaviour of operating systems with space and time partitioning.

To demonstrate the usefulness of this tool, a new portable and open source benchmark was developed especially for this type of operating systems. The benchmark is self-standing, but can optionally be integrated with the proposed trace analysis framework. By using this option, the developed view can be automatically synchronized to the occurrences of outlier values to observe the state of the system at these points. We will show how this functionality can be used to detect isolation problems while running applications concurrently.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

AADL	Architecture Analysis and Design Language
APEX	Application executive
API	Application programming interface
BTF	Best Trace Format
CPU	Central processing unit
CTF	Common Trace Format
DMA	Direct memory access
FPGA	Field-programmable gate array
GDB	GNU Debugger
IMA	Integrated modular avionics
I/O	Input/output
ISR	Interrupt service routine
kprobe	Kernel probe
LTTng	Linux Tracing Toolkit next generation
MMU	Memory management unit
Tcl	Tool Command Language
TSDL	Trace Stream Description Language
WCET	Worst case execution time
XML	Extensible Markup Language

CHAPITRE 1 INTRODUCTION

L'informatique embarquée est présente dans une vaste majorité des systèmes numériques modernes. Ces systèmes, développés pour accomplir des tâches bien précises, doivent généralement composer avec des contraintes temporelles, d'espace et énergétiques très strictes. De plus, dans certains domaines comme l'avionique, des contraintes de sûreté de fonctionnement supplémentaires s'appliquent. Le processus de développement se doit alors d'être des plus rigoureux afin de créer le système le plus performant possible respectant ces obligations. Puisque ce type de système s'intègre usuellement dans une chaîne de traitement répondant à des stimuli externes, leur débogage doit être possible sans la violation des contraintes temporelles. Le traçage du système pour l'analyse a posteriori de l'exécution devient alors une méthode intéressante pour l'évaluation des performances et le diagnostic de problèmes. Ce mémoire explorera donc l'utilité du traçage pour l'analyse de performance de système temps réels avec partitionnement spatial et temporel utilisé dans le domaine de l'avionique.

1.1 Définitions et concepts de base

Dans cette section, nous définissons certains concepts auxquels nous ferons référence à plusieurs reprises dans la suite du mémoire.

1.1.1 Système temps réel

Un système temps réel est caractérisé par sa capacité à traiter des évènements provenant de son environnement dans un délai maximum. Selon le domaine d'application, ce délai varie de l'ordre de la microseconde jusqu'à l'ordre de la seconde. Par conséquent, le critère du temps réel ne correspond pas à la vitesse d'exécution du système, mais bien au déterministe avec lequel il réussit à accomplir ses tâches. Pour certains domaines, la violation des contraintes temporelles n'entraîne pas de conséquences graves. Par exemple, un serveur de jeux en réseau se doit de répondre dans un délai maximum acceptable pour que les joueurs n'observent pas de ralentissements, mais un dépassement occasionnel du temps de réponse maximal voulu n'engendre pas de problèmes majeurs. Ces systèmes sont alors qualifiés de temps réel mou ou souple (*soft real time*). À l'inverse, d'autres domaines ne tolèrent aucune violation des contraintes temporelles. Par exemple, le système de freinage d'un véhicule intelligent doit absolument répondre dans les délais prévus. Ces systèmes sont alors classifiés comme temps réel dur ou critique (*hard real time*). Dans ce travail, nous nous intéressons à ce second type de

systèmes temps réel. Les prochaines mentions de ce concept font alors référence aux systèmes temps réel dur ou critique.

1.1.2 Partitionnement

Le partitionnement est la division d'une ressource en sections qui seront attribuées à des composantes d'un système. En informatique, ce concept est souvent appliqué au partitionnement de disque dur afin de délimiter des zones réservées à un usage en particulier. Il n'y est par contre pas limité. Le partitionnement peut être généralisé pour s'appliquer à toutes les ressources utilisables par un système informatique. Dans ce travail, nous explorerons l'utilité du partitionnement spatial et temporel, donc de la division du temps d'exécution et de la mémoire disponibles aux applications temps réel.

1.1.3 Ordonnancement

L'ordonnancement est le mécanisme par lequel le temps est réparti entre les différentes tâches du système. Une tâche est le contexte d'exécution du code relatif à une application. Une application peut être composée de plusieurs tâches et le système peut être appelé à gérer différentes applications en parallèle. Pour prendre une décision sur quelles tâches doivent s'exécuter à un temps donné, l'ordonnanceur suit la politique d'ordonnancement qui lui est attribuée. Dans un système temps réel multitâche, une politique préemptive basée sur les priorités est généralement utilisée. La préemption permet à une tâche de priorité plus élevée de remplacer une tâche de priorité plus faible déjà en cours d'exécution. La préemption est initiée par le système d'exploitation. Ce dernier est donc responsable de l'ordonnancement des tâches sur le système.

1.1.4 Traçage

Le traçage sert à récupérer de l'information à propos de l'exécution de programmes. L'information collectée est ensuite analysée pour fournir un rapport détaillé de l'exécution au développeur. Cette information provient de certains points spécifiques de l'application ou du système d'exploitation qui ont été instrumentés. Le type et la qualité de l'information varient donc selon les efforts mis en place pour l'instrumentation du code. Durant l'exécution, une application ou une partie du système d'exploitation appelée le traceur s'occupe d'enregistrer l'information désirée sur un support permanent qui pourra être lu après l'exécution.

Évènement

Un évènement est l'occurrence d'une action d'intérêt dans le système tracé. Il est donc associé à un temps précis durant l'exécution et à un type. Selon le type, l'évènement contient des valeurs qui permettent de spécifier le contexte de son occurrence. Un évènement est créé lorsqu'un point de trace est rencontré durant l'exécution.

Point de trace

Le point de trace est l'endroit où l'instrumentation est placée afin d'enregistrer un évènement. Selon le traceur utilisé, il peut être placé dans du code usager ou du code du système d'exploitation. Les points de trace sont généralement optionnels et peuvent être activés ou désactivés. La désactivation permet d'éviter le surcoût inévitable du traceur qui doit enregistrer l'information créée par le point de trace lorsqu'il est rencontré.

1.1.5 Banc d'essai

Un banc d'essai est une suite de tests standardisés visant à comparer les performances d'un système précis, par exemple un processeur, un algorithme ou un système d'exploitation. Pour que le banc d'essai soit utile, ses résultats se doivent d'être facilement interprétables et représentatifs de cas d'utilisation réels.

1.2 Éléments de la problématique

Les systèmes temps réel dur sont soumis à des contraintes temporelles, spatiales et de sureté très strictes. De plus, l'utilisation de système avec partitionnement spatial et temporel, permettant la cohabitation d'applications de différents niveaux de criticité, augmente la complexité de ces systèmes. En effet, avec l'accessibilité grandissante de processeurs multicoeurs pour l'informatique embarquée temps réel, il est devenu envisageable de laisser une application temps réel souple partager le même processeur qu'une application temps réel dur. Toutefois, cette cohabitation est seulement possible lorsqu'un système d'exploitation permettant un partitionnement spatial et temporel robuste est utilisé. Le système d'exploitation devra être conçu de façon à éviter des problèmes d'interférence entre les applications, non existants pour des processeurs à coeur unique. Si ces interférences sont mal contenues par le système d'exploitation, nous pourrions observer des temps d'exécution plus longs que prévu pour une application critique, lorsqu'elle est exécutée côte à côte avec une autre application. Ces problèmes seront difficilement reproductibles et diagnosticable. Puisque les développeurs

d'application visant ce type de système ne sont pas à l'abri de problèmes de performance du système d'exploitation lui-même, ou de problèmes dans des applications elles-mêmes parallélisées, la disponibilité d'outils appropriés devient un enjeu important.

Dans un contexte temps réel, l'efficacité d'un débogueur traditionnel est limitée. L'arrêt complet de l'exécution sur un coeur risque de causer le dépassement d'échéance de réponse dans l'application analysée ou même dans d'autres applications concurrentes qui ne pourront pas s'exécuter sur le coeur bloqué. De plus, si le problème observé survient plus tard dans la chaîne de traitement, il est impossible d'observer le flot normal d'exécution de l'application afin d'identifier la source du problème. La mise en place de solutions beaucoup moins intrusives doit donc être envisagée.

Le traçage logiciel du système d'exploitation devient alors un outil intéressant pour le diagnostic de problèmes provenant de contextes hautement parallèles. En utilisant un traceur ne causant qu'un surcoût minimal et prédictible sur l'exécution, il est possible d'extraire une trace d'exécution comprenant l'information nécessaire à la reconstruction d'un historique d'états des applications du système. En limitant la quantité d'évènements produite à un minimum, le comportement des applications du système sera le même qu'en temps normal. Pour être encore plus efficaces dans le contexte d'un système d'exploitation avec partitionnement spatial et temporel, l'analyse et la visualisation des traces devront toutefois être adaptées à leur réalité. Le partitionnement utilisé est plus strict que celui qui est possible à travers l'utilisation de machines virtuelles ou de conteneurs déjà étudiés dans [1] et [2]. Dans le contexte qui nous intéresse que décrit la section 2.1, il est important de pouvoir repérer facilement le dépassement de l'utilisation permise de ces ressources, en plus de permettre la visualisation de l'état des applications et des ressources utilisées par celles-ci.

Une autre source de questionnement pour la préparation de projets d'informatique embarquée est le fractionnement de l'offre dans les systèmes d'exploitation temps réel. Plusieurs compagnies rivales tentent de prouver que leur produit est le plus rapide et efficace. Chacune d'entre elles a avantage à présenter son produit sous son meilleur jour. Prendre une décision éclairée peut s'avérer difficile si les résultats de performance fournis par les manufacturiers sont issus de systèmes et de configurations différents. Le développement d'un banc d'essai ouvert permettant l'obtention de résultats fiables et tangibles faciliterait la préparation d'analyses comparatives de systèmes d'exploitation temps réel. Pourtant, l'offre actuelle de banc d'essai crédible et portable sur différents systèmes d'exploitation est quasi inexistante. L'utilité d'un tel banc d'essai serait double. Il permettrait la caractérisation de performances et pourrait servir de charge de travail afin de diagnostiquer les éventuels problèmes à l'aide du traçage.

1.3 Objectifs de recherche

Les objectifs de recherche poursuivis sont donc les suivants :

1. Développer un environnement d'analyse de traces de systèmes d'exploitation avec partitionnement spatial et temporel
2. Développer un banc d'essai permettant de caractériser les performances d'un système d'exploitation avec partitionnement spatial et temporel.
3. Réaliser une étude comparative de performances de systèmes d'exploitation en se basant sur le banc d'essai préparé.
4. Valider que l'environnement d'analyse de traces construit permet le diagnostic de problèmes de performance lors de l'analyse des résultats de l'étude comparative.

1.4 Plan du mémoire

Dans le chapitre 2, une revue de littérature présente l'état de l'art dans les domaines des systèmes temps réels avec partitionnement spatial et temporel, du traçage temps réel et de bancs d'essai permettant la caractérisation de performances des systèmes d'exploitation. Le chapitre 3 fait état de la méthodologie utilisée afin d'accomplir les objectifs de l'étude. Puis, le chapitre 4 contient un article scientifique décrivant l'implémentation de la solution proposée ainsi que les résultats de l'analyse comparative de performances de systèmes d'exploitation. Le chapitre 5 approfondit certains résultats du chapitre précédent. Nous conclurons dans le chapitre 6 en discutant de possibles améliorations futures à la solution proposée.

CHAPITRE 2 REVUE DE LITTÉRATURE

Dans ce chapitre, nous définissons les caractéristiques des systèmes temps réel avec partitionnement spatial et temporel et présentons les projets à code source libre respectant celles-ci, puis nous discutons des défis posés par ces particularités. Nous terminons le chapitre en étudiant les outils de traçage disponibles pour ce type de système et les banc d'essai existants permettant d'en caractériser la performance.

2.1 Systèmes temps réel avec partitionnement spatial et temporel

Au cours des vingts dernières années, l'industrie avionique s'est distancée des architectures fédérées pour se tourner vers un nouveau paradigme architectural : l'avionique modulaire intégrée (IMA). Plutôt que de dupliquer les ressources pour chacune des applications du système, celles-ci se partagent les ressources [3]. Ce virage est motivé par l'augmentation de la complexité des systèmes et, par conséquent, de la quantité de ressources nécessaires. Dans une architecture fédérée, le partage des ressources est très limité. Ceci entraîne une augmentation des coûts de production, de la masse et de la consommation énergétique du système [4].

Pour que la cohabitation entre les applications soit possible, le partage des ressources doit être encadré par un méchanisme déterministe permettant d'assurer la disponibilité des ressources nécessaires au fonctionnement des applications. Ce contexte a mené au développement des systèmes d'exploitation temps réel avec partitionnement spatial et temporel [5]. Dans le domaine de l'avionique commerciale, la norme ARINC 653 [6] a été établie afin de régir le comportement de tels systèmes d'exploitation dans le contexte de l'avionique modulaire intégrée.

Avant de détailler le comportement d'un système d'exploitation basé sur la norme ARINC 653, voyons comment il s'intègre dans une architecture avionique modulaire intégrée. Selon ce mode de conception, les sous-systèmes assemblés pour former un système avionique sont séparés en modules. Un module contient des applications logicielles et les ressources matérielles utilisées par celles-ci. Le module est donc le bloc de plus haut niveau du système. À l'intérieur du module, nous trouvons le système d'exploitation avec partitionnement spatial et temporel qui orchestre la cohabitation des applications. Selon la norme ARINC 653, une application est composée d'une à plusieurs partitions logicielles. Les partitions sont le contexte d'exécution des processus qui accomplissent les tâches de l'application. Ces processus interagissent

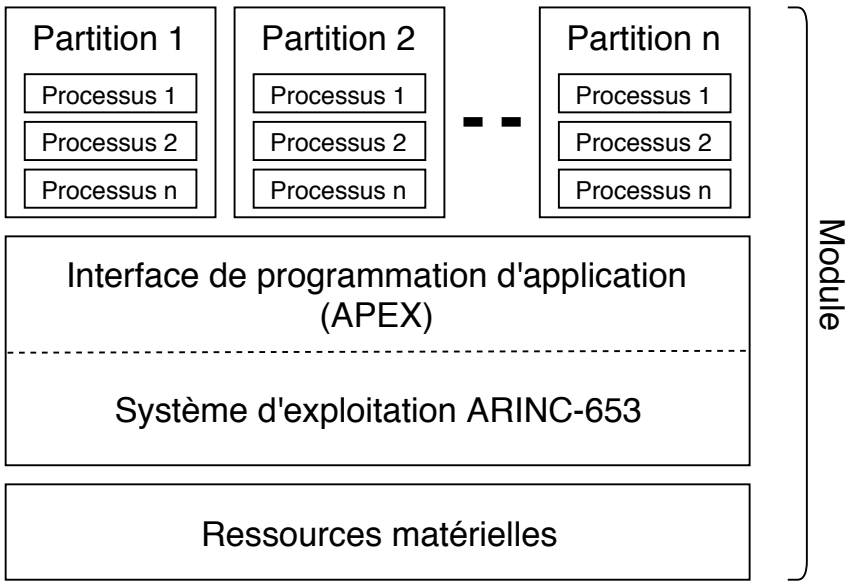


Figure 2.1 Structure d'un module d'avionique modulaire intégrée.

avec le système d'exploitation à travers une interface de programmation d'application (API) définie par la norme ARINC 653. Cette interface est nommée *Application Executive* (APEX). La figure 2.1 illustre la relation hiérarchique entre les composantes d'un module.

Le comportement d'un système d'exploitation avec partitionnement spatial et temporel est guidé par ces deux garanties qu'il doit fournir. Dans le cadre défini par le standard ARINC 653, le partitionnement temporel est assuré par un ordonnancement statique et déterministe des partitions [7]. Une plage de temps fixe qui se répète périodiquement est assignée au module (*major time frame*). La plage de temps globale est subdivisée en sous-plages d'exécution, chacune attribuée à une partition. Les sous-plages d'exécution d'une partition sont définies par un décalage par rapport à la plage d'exécution globale et par une durée. La plage d'exécution globale peut inclure des sous-plages non utilisées, mais chaque partition définie doit y être exécutée un minimum d'une fois. La figure 2.2 montre un exemple d'ordonnancement possible pour trois partitions s'exécutant à l'intérieur d'une plage globale de 100 millisecondes. Lorsqu'un processeur multicoeur est utilisé, un attribut de la partition indique les coeurs qui lui sont accessibles. Par conséquent, une partition peut occuper tous les coeurs disponibles ou plusieurs partitions peuvent s'exécuter en parallèle [8]. Un processus à l'intérieur d'une partition peut aussi définir une affinité vers un groupe de coeurs afin de limiter son exécution à ceux-ci. L'ordonnancement des processus s'effectue localement à l'intérieur d'une partition. Le processus ayant la plus haute priorité et étant prêt à s'exécuter est celui qui est choisi pour l'exécution, tout en respectant les affinités. La priorité des processus est définie lorsqu'ils sont créés au moyen de l'interface de programmation d'application du

système d'exploitation.

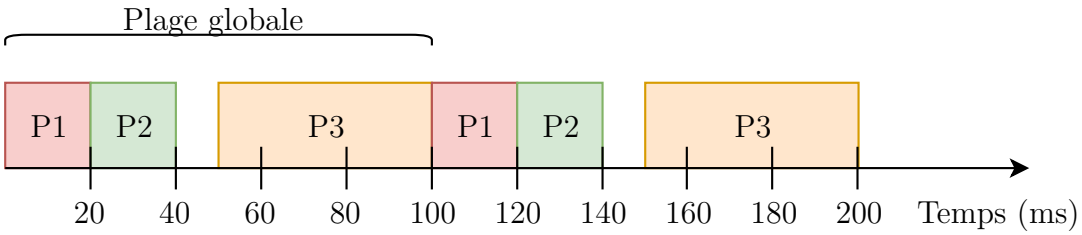


Figure 2.2 Ordonnancement possible de trois partitions.

Le partitionnement spatial découle de la division des applications en partitions. Chaque partition exécute les processus dans un environnement contrôlé et isolé des autres partitions du module. L'utilisation d'une unité de gestion de la mémoire est la façon la plus commune de créer cet environnement [9]. Cette unité accompagne la vaste majorité des processeurs suffisamment puissants pour partager des ressources entre plusieurs applications. Le fonctionnement exact de cette unité varie d'une architecture à l'autre, mais l'essence reste la même. Les adresses manipulées par le processeur sont vérifiées et transformées à partir de tables configurées par le système d'exploitation, et lues par le circuit matériel de l'unité de gestion mémoire, afin que cette opération se fasse rapidement. Le noyau peut donc initialiser ces tables de façon à ce que les processus ne manipulent que des adresses pour lesquelles les permissions de la partition sont suffisantes. Dans le cas d'un module utilisant un processeur multicoeur, il est nécessaire de considérer les effets du partage de la cache. En effet, deux partitions concurrentes s'exécutant sur des coeurs différents peuvent partager un ou plusieurs niveaux de caches (L2 et L3), ce qui risque de causer une interférence matérielle [10]. Cette interférence peut être limitée par l'utilisation du verrouillage de cache et du partitionnement de cache [11]. Le verrouillage de cache permet d'empêcher que des données chargées en mémoire ne soient évincées durant l'exécution. Le verrouillage peut être fait statiquement, avant le début de l'exécution, ou dynamiquement durant l'exécution. Le partitionnement de cache est l'assignation d'une partie de la cache à une partition ou un coeur. Les deux méthodes s'emploient de concert afin d'éviter les effets de l'interférence. Par exemple, une solution pour un module utilisant un processeur à deux coeurs partageant la cache L2 serait de diviser la mémoire L2 en deux sections continues (partitionnement de cache), puis d'installer les données d'une partition lorsqu'elle est ordonnancée et de seulement les évincer lorsque la partition suivante commence son exécution (verrouillage de cache). Le standard ARINC 653 est un guide pour implémenter un système d'exploitation avec partitionnement spatial et temporel. Puisque ces systèmes sont utilisés dans des contextes très spécialisés, très peu d'implémentations de ceux-ci sont rendues disponibles sous une licence libre. Dans les

prochaines sous-sections, nous présentons les projets qui ont une implémentation compatible avec le partitionnement spatial et temporel ou s'en approchant.

2.1.1 POK Kernel

POK est un système d'exploitation temps réel fournissant des garanties d'isolation spatiale et temporelle [12]. POK implémente le partitionnement spatial et temporel de la même façon que la norme ARINC 653 le suggère. Le système d'exploitation gère des partitions qui sont ordonnancées de manière fixe et périodique. Les partitions sont isolées spatialement grâce à l'utilisation de l'unité de gestion de la mémoire. L'ordonnancement des tâches à l'intérieur d'une partition est géré par le même noyau. Une particularité de ce système d'exploitation est qu'il est compatible avec un mode de développement par modèle avec le *Architecture Analysis and Design Language* (AADL). La configuration des partitions ainsi que le code d'initialisation de celles-ci peuvent donc être générés automatiquement à partir du modèle du système désiré.

Ce système d'exploitation est compatible avec les architectures x86, PowerPC et SPARC. Toutefois, aucune information n'est disponible à propos des systèmes sur puce avec lesquels il a été testé. De plus, le projet ne fait pas l'objet de mises à jour régulières. Le système d'exploitation JetOS est un dérivé de POK. Ce dernier système d'exploitation fonctionne de la même façon, mais certaines fonctionnalités ont été modifiées. Entre autres, l'ordonnanceur de processus a été complètement réécrit afin de correspondre exactement à celui décrit dans le norme ARINC 653 [13].

2.1.2 Linux

Linux est un système d'exploitation hautement configurable qui offre un support partiel pour les applications temps réels. Afin de limiter les latences observées au maximum, le correctif *PREEMPT_RT* devrait être appliqué au noyau utilisé pour ordonnancer des applications temps réels. Avec les années, une grande partie des fonctionnalités seulement rendues disponibles par ce correctif ont été intégrées au noyau de base. Toutefois, la seule façon de permettre la préemption dans toutes les sections du noyau est d'appliquer ce correctif [14]. Les tâches créées sur Linux peuvent changer leur politique d'ordonnancement pour une politique temps réel comme *SCHED_FIFO*, ce qui assure qu'elles seront exécutées avant les autres tâches. Cette politique correspond à un ordonnancement préemptif par priorité où les tâches sont seulement suspendues si elles attendent l'arrivée d'un évènement, ou si une tâche de plus haute priorité est prête.

Bien que Linux ne soit pas développé avec l'objectif d'être un système d'exploitation avec partitionnement spatial et temporel, il offre des services qui permettent de restreindre les ressources temporelles et spatiales disponibles aux applications. Premièrement, les groupes de contrôles peuvent être utilisés pour limiter de façon stricte les ressources utilisées par un groupe de tâches. Par exemple, il est possible de définir des bornes maximales sur la quantité de mémoire allouée, la bande passante du disque utilisée et le temps d'exécution [15]. Le temps d'exécution maximal est spécifié par période d'exécution, tout comme le suggère la norme ARINC 653. Par contre, il n'est pas possible de définir explicitement un ordre d'exécution entre les groupes de contrôles. Deuxièmement, l'isolation de processeurs permet de filtrer quelles tâches peuvent s'exécuter sur un groupe de coeurs. Lorsqu'un coeur est isolé, les seules tâches qui peuvent s'y exécuter sont celles qui y sont délibérément placées. En combinant ces deux mécanismes, nous pouvons envisager une configuration de groupes de contrôle qui permettrait d'ordonnancer les tâches de manière similaire à un système d'exploitation avec partitionnement spatial et temporel.

2.1.3 Xen

Xen est un hyperviseur natif utilisé pour l'orchestration de machines virtuelles paravirtualisées [16]. La paravirtualisation consiste à modifier les systèmes d'exploitation utilisés comme machines virtuelles, pour une exécution dans un contexte de virtualisation plutôt que de forcer l'orchestrateur à émuler entièrement les ressources matérielles. Xen fonctionne avec le concept de domaine. Le domaine est le contenant d'une machine virtuelle pour lequel des paramètres comme le nombre de coeurs disponibles ou la mémoire disponible sont attribués. Pour configurer le système, un domaine privilégié appelé le domaine 0 (dom0) est utilisé.

L'utilisation d'une machine virtuelle fournit déjà la l'isolation spatiale requise pour le partitionnement spatial et temporel. Pour isoler temporellement les domaines, un ordonnanceur compatible avec la norme ARINC 653 a été introduit dans Xen [17]. Lorsque cet ordonnanceur est utilisé pour un groupe de domaines, chacun d'entre eux se voit assigné un temps d'exécution fixe qui s'inscrit dans une plage d'exécution globale. Toutefois, les domaines ordonnancés avec cette politique ne peuvent utiliser plus d'un coeur.

L'emploi d'un hyperviseur pour atteindre le partitionnement spatial et temporel se distingue de POK ou Linux. En effet, ces derniers sont des systèmes d'exploitation traditionnels qui implémentent le partitionnement entre les applications sans pour autant permettre l'exécution d'un autre système d'exploitation à l'intérieur des partitions.

2.1.4 XtratuM

XtratuM est un second hyperviseur natif possédant une politique d'ordonnancement conforme à la norme ARINC 653. Contrairement à Xen, XtratuM est conçu spécialement avec la norme ARINC 653 en tête [18]. Par conséquent, la configuration des partitions gérées par l'hyperviseur se fait entièrement à l'aide de fichiers de configuration XML, comme le suggère la norme ARINC 653. Puisqu'il s'agit d'un hyperviseur, XtratuM ne s'occupe pas de l'ordonnancement des tâches à l'intérieur des partitions. Pour fournir un environnement de développement complet conforme avec la norme ARINC 653, les partitions doivent exécuter des systèmes d'exploitation temps réel respectant cette norme. Le code source de XtratuM est disponible sur le site web de la compagnie FentISS qui en gère le développement.

2.2 Problèmes dans les systèmes embarqués temps réel avec partitionnement spatial et temporel

Cette section passe en revue les problèmes potentiels émanant de l'utilisation d'un système d'exploitation temps réel avec partitionnement spatial et temporel sur des processeurs multicoeurs. Nous dégageons trois catégories de problèmes : la détermination du pire cas de temps d'exécution, la gestion des erreurs durant l'exécution et le partage de ressources matérielles autres que le processeur.

2.2.1 Pire cas de temps d'exécution

Le pire cas de temps d'exécution (Worst case execution time (WCET)) est le temps maximum qu'une partie d'un programme peut mettre à s'exécuter sur un système donné. Cette borne est entre autres utile pour effectuer une étude de faisabilité d'ordonnancement des tâches sur un système temps réel [19]. Une grande variété de méthodes peut être utilisée afin de dériver ce temps maximum. Généralement, les méthodes de détermination du WCET sont divisées entre méthodes statiques et méthodes dynamiques ou basées sur des mesures. Les méthodes statiques sont basées sur une analyse statique du code source, potentiellement annoté, ou compilé d'une tâche. Les résultats de cette analyse, combinés à un modèle mathématique précis du processeur ciblé, permettent de calculer une borne supérieure du temps d'exécution. Les méthodes dynamiques peuvent quant à elles aussi se baser sur une première étape d'analyse statique, mais utilisent une série de simulations ou d'exécutions sur le processeur ciblé plutôt qu'un modèle de celui-ci pour le calcul. Cette séparation des méthodes d'analyse perd un peu de son intérêt avec la perte d'efficacité des méthodes statiques lors de l'utilisation de processeurs modernes ayant plusieurs niveaux de caches et des unités arithmétiques

et logiques en parallèle [20]. Au cours de la dernière décennie, l'intérêt de recherche s'est porté vers les méthodes probabilistes d'estimation de pire cas de temps d'exécution. Ainsi, la décomposition en méthodes d'analyse déterministes et méthodes d'analyse probabilistes proposée par [21] dépeint mieux les types de méthodes composant actuellement l'état de l'art.

Méthodes déterministes Les méthodes déterministes produisent une valeur discrète comme résultat. Pour que la méthode soit jugée utile, il faut que cette borne soit effectivement impossible à franchir. Ces méthodes sont plus matures et implémentées dans plusieurs outils commerciaux [22].

Méthodes probabilistes Les méthodes probabilistes donnent une distribution de probabilité de la borne supérieure du temps d'exécution. Le développeur choisit pour les analyses subséquentes une borne qui est en accord avec les requis de fiabilité du système. Ce genre méthode est plus récent, mais a déjà été appliqué avec succès dans le domaine de l'avionique. [23]

Comme mentionné plus tôt, les méthodes statiques déterministes deviennent de plus en plus coûteuses à déployer pour des architectures de processeurs modernes. Les résultats sont sécuritaires, mais trop pessimistes. L'auteur de [24] rapporte cinq obstacles majeurs à l'utilisation d'une méthode d'analyse statique dans une application embarquée de calibre industrielle de contrôle de satellite .

— Un programme complexe combiné avec un domaine d'état possible très large rend la tâche computationnelle difficile et laborieuse lorsque l'usager doit l'exécuter plusieurs fois afin de raffiner le résultat avec des annotations.

— L'annotation manuelle du programme repose sur l'hypothèse que les annotations des développeurs seront exactes.

— Les outils ont de la difficulté à analyser du code utilisant certain patrons d'usage comme l'allocation dynamique de mémoire ou les pointeurs vers des fonctions qui ne peuvent pas être éliminés facilement de code déjà existant.

— L'utilisation d'interface générique réutilisable entre les projets peut cacher de l'information à l'outil d'analyse.

— L'analyse statique ne considère pas des interactions entre les tâches.

Les méthodes déterministes basées sur des mesures décomposent le code étudié en sous-sections pour lesquelles le temps maximal d'exécution est déterminé. Ces sous-sections du programme peuvent être exécutées sur le processeur ou dans un simulateur avec différents paramètres afin d'en mesurer le temps réel pris. L'exécution permet d'éviter l'utilisation d'un

modèle mathématique précis du processeur pour l'analyse. Dans [25], les auteurs soulignent que la méthode utilisée doit effectivement borner supérieurement le pire temps d'exécution et ne pas considérer le temps maximal observé comme cette borne. Ce serait seulement le cas si un test exhaustif sur tous les états du processeur et toutes les valeurs d'entrées possibles avait été effectué.

Les méthodes probabilistes peuvent aussi être statiques ou basées sur des mesures. Une méthode probabiliste statique nécessite tout de même un modèle détaillé du processeur ciblé. Par exemple, la méthode proposée dans [26] prend en entrée une distribution de probabilité pour le temps pris par chacune des instructions exécutables sur processeur, ou du moins celles contenues dans le programme étudié. Ce genre de modèle n'est pas toujours disponible, donc des méthodes probabilistes basées sur la mesure ont aussi été proposées. Une telle méthode basée sur la théorie statistique des mesures extrêmes est proposée dans [27]. Leur méthode est plus pessimiste de 2 à 15% qu'une méthode statistique probabiliste, mais évite l'utilisation d'un modèle détaillé du temps pris par chaque instruction par le processeur.

L'efficacité de ces méthodes reste limitée lors de l'utilisation de processeur multicoeur standard (*commercial off-the-shelf* (COTS)). Le partage de la cache, des bus d'accès à la mémoire, des pipelines d'exécution lors de l'utilisation de *simultaneous multithreading* (Hyper-Threading chez Intel) ou encore des périphériques adressables entraînent des interférences matérielles affectant le déterminisme des opérations exécutées. Ainsi, la borne résultante de l'analyse sera plus pessimiste que sur un système à coeur unique afin de tenir compte de latences imprévues [28]. Une option pour améliorer la qualité des bornes est d'opter pour des processeurs multicoeurs spécialisés pour les domaines temps réel. Par exemple, la série AURIX TC27x de la compagnie Infineon offre une architecture multicoeur plus facile à modéliser pour une analyse de pire cas de temps d'exécution [29]. Les éléments architecturaux la rendant plus déterministe sont le bus d'accès à la mémoire avec redondance sous forme de réseau crossbar et les caches locales à chaque coeur sans mécanisme de cohérence. La redondance du bus permet de borner les temps d'accès au bus, puisqu'il ne peut pas être monopolisé par un autre coeur. Les caches exclusives à chaque coeur évitent que des applications concurrentes compétitionnent pour l'espace en cache.

Malgré les limitations de l'analyse de pire cas de temps d'exécution pour les processeurs multicoeurs, l'utilisation de cette catégorie de processeur reste attrayante pour l'avionique modulaire intégrée [30]. Choisir des bornes pessimistes ne permet pas une utilisation maximale des ressources. Afin d'obtenir une performance maximale, il peut être tentant d'expérimenter avec des bornes plus strictes durant le développement du système. Il faut alors être conscient qu'un calcul initial erroné du pire temps d'exécution peut être la cause directe d'une échéance

manquée ou encore affecter les échéances d'autres tâches de plus basses priorités [31]. Dans ce genre de situation, il est essentiel de rendre accessible des outils permettant de diagnostiquer et comprendre la source de ce type de problème indirect et difficile à prédire à priori.

2.2.2 Gestion des erreurs

Un des objectifs motivant l'isolation spatiale et temporelle est la cohabitation d'application avec différents niveaux de criticité. L'application ayant le plus haut niveau de criticité sur le système doit avoir la garantie qu'une erreur rencontrée dans une application contenue dans une partition distincte n'aura aucun impact sur son exécution [32]. Cette garantie est fournie par la norme ARINC 653 avec une approche à deux volets : l'isolation des erreurs par le partitionnement et le recouvrement des erreurs lors de l'exécution. Il est important de noter que cette stratégie ne vise que les erreurs logicielles et non la récupération et l'isolation d'erreurs causées par du matériel fautif.

Isolation des erreurs

L'isolation d'erreurs découle naturellement du partitionnement spatial et temporel forcé par la norme ARINC 653. L'implémentation de ces deux concepts permet de garantir qu'une erreur à l'intérieur d'une partition ne causera pas la corruption ou le vol de temps d'une autre partition. Toutefois, pour que le système soit fiable en plus d'être stable, le recouvrement des erreurs doit être possible. La norme ARINC 653 définit un module de surveillance de la santé du système (*Health Monitoring*) pour accomplir cette tâche.

Recouvrement d'erreurs

Les mécanismes de recouvrement d'erreurs proposés par la norme ARINC 653 sont regroupés dans une fonctionnalité du système d'exploitation nommée le *Health Monitor* ou moniteur de santé. Le moniteur de santé peut prendre certaines actions prédéfinies selon le type d'erreur rencontrée. Les erreurs sont classifiées selon trois niveaux [33].

Niveau des processus Ces erreurs touchent d'un à plusieurs processus contenus à l'intérieur d'une même partition. Des exemples d'erreurs de ce type sont des accès à de la mémoire non autorisée, des violations d'échéances et des appels invalides à des services du système d'exploitation.

Niveau des partitions Ces erreurs affectent une partition en particulier. Des exemples de ce type d'erreur seraient des tables de configuration invalides, une exception lancée par

le processus de gestion d'erreur local à la partition ou encore un problème rencontré lors de l'initialisation.

Niveau du module Ces erreurs sont globales et risquent d'affecter toutes les partitions. Des exemples de ce type d'erreur sont un problème d'alimentation, une erreur lors de la restauration du contexte d'une partition ou une erreur d'initialisation du module.

Les actions à prendre pour des erreurs au niveau du module ou des partitions sont configurables à travers des tables de configurations. La table de configuration du module définit l'action à prendre lorsqu'une erreur au niveau du module survient, si elle n'est pas ignorée. Les actions possibles sont d'éteindre le module, le redémarrer ou d'entamer une action de recouvrement spécifique au système d'exploitation. De façon analogue, la table de configuration d'erreurs de partition spécifie le comportement du système d'exploitation pour gérer ce type d'erreur si elle n'est pas ignorée. Les actions possibles sont d'arrêter la partition, de redémarrer la partition ou d'entreprendre une action spécifique au système d'exploitation. Les erreurs au niveau des processus sont gérées différemment. Chaque partition peut créer un processus spécial ayant la plus haute priorité de la partition qui sera réveillé lorsque le système d'exploitation détectera une erreur. Le processus gestionnaire d'erreur peut alors ignorer l'erreur, redémarrer le processus fautif, arrêter la partition ou la rédémarrer.

Les erreurs sont détectées automatiquement par le système d'exploitation (interruption matérielle) ou relevées par une application par le biais de l'interface APEX. Les erreurs créées à travers cette interface sont gérées par le processus spécial dédié à la gestion des erreurs. Ce mécanisme de notifications permet la mise en place de solutions de surveillance et de recouvrement d'erreur beaucoup plus sophistiquées que celles minimalement demandées par le standard ARINC 653 [34].

Dans un premier effort, les auteurs de [35] proposent de définir un modèle des propriétés du système basé sur une machine à état fini. La machine à état est stimulée de façon périodique par un processus moniteur instancié à l'intérieur de chacune des partitions. Pour que le moniteur découvre les états fautifs, la modélisation des propriétés surveillées doit contenir les transitions menant à un état fautif. Lorsqu'une erreur est levée, le processus gestionnaire d'erreur est réveillé par le mécanisme de notifications implémenté par le système d'exploitation. Pour que cette approche soit fonctionnelle, elle doit être implémentée dès le début du développement, puisqu'elle nécessite l'ajout de processus supplémentaires et l'implémentation de communications entre le processus moniteur et les processus de travail. La méthode proposée permet de limiter les dégâts en cas de fautes imprévues, mais ne permet pas le diagnostic de celles-ci.

Une approche plus raffinée du moniteur de propriété dans le système est proposée dans [36].

Leur méthode est basée sur l'utilisation du modèle par composant ARINC (ACM) pour le développement. En suivant ce modèle, les sous-parties de l'application sont divisées en composantes ayant des interfaces de communication définies formellement. Le développeur peut ajouter à son modèle des moniteurs de propriétés sur les interfaces de communication qui rapportent les problèmes à un processus de gestion de la santé (*Health Manager*) local à chaque composante. Le processus de gestion de la santé contient une machine à état fini qui évolue selon les évènements reçus par les moniteurs. Ce processus a deux rôles. Il prend les actions nécessaires pour régler les erreurs et notifie l'engin d'agrégation des erreurs des actions encourues. L'engin d'agrégation communique à son tour avec l'engin de diagnostic qui produit des hypothèses sur les causes probables des erreurs. Ces deux engins peuvent être déployés dans une partition sur le même module ou dans un module différent. L'approche proposée est intrusive puisqu'elle demande la création d'un nouveau processus par composant et d'au minimum une nouvelle partition. Cette solution de monitorage donne des hypothèses sur les sources d'erreurs, mais ces hypothèses sont limitées par l'efficacité des algorithmes utilisés.

Les méthodes proposées par le standard ARINC 653 pour la récupération d'erreur sont utiles et efficaces pour éviter un malfonctionnement de l'application. Nous dénotons un besoin d'augmenter ces capacités afin d'implémenter une surveillance en continu du système. Par contre, les méthodes proposées jusqu'à maintenant pour la surveillance en continu demandent d'insérer de nouvelles composantes dans le système, ce qui n'est pas désirable pour des systèmes critiques demandant une phase de certification puisque ces composantes devraient elles aussi être certifiées.

2.2.3 Partage des ressources matérielles

Une source de problèmes dans tout système sur lequel cohabitent plusieurs applications est le partage des ressources matérielles. Le partitionnement temporel complique ce défi puisque les opérations entamées par une partition avant sa suspension ne doivent pas impacter les partitions suivantes.

Un accès direct à la mémoire (DMA) fait en fin de plage de temps d'une partition bloquerait la disponibilité de cette ressource pour la partition suivante. Ce problème peut être contourné en interdisant l'accès au contrôleur DMA pour toutes les partitions. Une seconde solution serait d'encapsuler les accès DMA dans une interface vérifiant que le temps d'exécution restant à la partition est suffisant pour l'accès [37]. Cette dernière méthode assume qu'on peut borner de façon sécuritaire le temps maximal nécessaire. Les accès I/O en général posent aussi le même problème. Toutefois, il serait beaucoup plus restrictif de les empêcher complètement sur le

système. Une solution envisageable est de limiter les accès I/O à une seule partition. Ainsi, les périphériques utilisés ne sont pas en situation de contention [38]. Cette solution limite le débit maximal possible du système selon la fréquence d'exécution de la partition gestionnaire des accès I/O. De plus, lorsqu'un accès I/O fait partie du chemin critique d'un processus temps réel, sa latence peut en être grandement affectée. Nous pouvons imaginer une situation où un processus attend l'arrivée d'un message sur une queue de message avec lequel une valeur doit être calculée puis écrite via un accès I/O. Si le message attendu arrive immédiatement après la fin du temps d'exécution de la partition, le message sera alors seulement lu lors de la prochaine exécution. Si l'accès I/O est traité en différé par une seconde partition, la latence est encore plus grande. Pour éviter ce problème dans l'implémentation d'un ordonnanceur ARINC 653 dans l'hyperviseur Xen, des queues de messages sont utilisées pour écrire dans un tampon circulaire unique à chaque partition [17]. Lorsque l'hyperviseur s'exécute, le tampon est lu par le pilote de périphérique. Toutefois, cette méthode force une interruption périodique durant l'exécution des partitions pour laisser l'hyperviseur s'exécuter.

La méthode la plus commune pour gérer les accès I/O reste l'ajout d'une partition dédiée [39]. Nous notons l'utilisation d'une partition dédiée aux accès I/O dans le développement d'un système de contrôle de mission spatiale décrit dans [40] et dans l'étude de cas du transfert d'un système de contrôle de mission aérospatiale vers un système avec partitionnement spatial et temporel présenté dans [41]. L'utilisation d'une partition dédiée permet aussi d'éviter la certification du pilote de périphérique au plus haut niveau de criticité présent sur le système, comme ce serait le cas si le pilote se retrouvait du côté du système d'exploitation. Cet avantage est non négligeable dans le domaine de l'avionique commerciale. Pourtant, l'utilisation d'une partition dédiée introduit des latences possibles dans les écritures et lectures, qui seront difficiles à diagnostiquer sans outils permettant d'observer l'état du système.

Les auteurs de [42] étudient les effets du partitionnement temporel sur la latence de communication entre deux modules distincts échangeant des messages sur un canal de communication partagé. Les auteurs montrent que, selon la latence causée par le canal de communication et la fréquence d'ordonnancement des partitions, certains messages peuvent être manqués. La solution proposée est le développement d'un outil de simulation permettant de déterminer les paramètres de partitionnement temporel optimaux. Cette solution est valide dans la mesure où il est possible de borner fidèlement, sans trop de pessimisme, les latences maximales et minimales du canal de communication utilisé.

2.3 Traçage logiciel

Dans la section précédente, nous avons introduit trois catégories de problèmes rencontrés lors du développement sur des systèmes temps réel avec partitionnement spatial et temporel. Nous avons identifié que ces problèmes peuvent engendrer des problèmes de performance difficiles à diagnostiquer sans l'aide d'outil approprié. Nous étudions maintenant l'utilisation du traçage et de l'analyse de traces en tant qu'outil de diagnostic de problèmes de performances.

Le traçage est une méthode de collecte d'informations à propos de l'exécution des tâches sur un système. Les informations recueillies par le traçage sont des évènements de bas niveau qui, une fois regroupés, forment une trace. La trace est ensuite analysée pour fournir une image à plus haut niveau de l'état du système à un moment donné, permettant de déterminer la source d'un problème [43]. Contrairement au débogage, le traçage ne nécessite pas l'arrêt de l'exécution de l'application afin d'en examiner l'état. Ceci en fait une méthode complémentaire très utile lors de l'investigation d'un problème d'échéance manquée qui ne pourrait pas être observée en arrêtant complètement l'application.

La collecte des évènements par le traceur passe par l'instrumentation du code, c'est-à-dire l'insertion de points de trace à des endroits précis afin d'en extraire une valeur ou un état. Deux options sont possibles pour l'instrumentation du code : l'instrumentation statique et dynamique [44]. L'instrumentation statique consiste à placer des points de trace dans le code de l'application avant la compilation. Cette méthode est plus performante, mais moins flexible puisque le code doit être compilé à nouveau pour ajouter un point de trace. L'instrumentation dynamique consiste à insérer des points de trace dans le code binaire de l'application durant son exécution. Cette façon de faire est plus flexible, mais engendre un surcoût plus élevé.

2.3.1 Traçage sous Linux

Pour un système d'exploitation largement répandu comme Linux, les solutions de traçage sont multiples. Elles ne sont par contre pas toutes adaptées à une utilisation dans un contexte temps réel. Les auteurs de [45] comparent les performances de différents traceurs Linux selon le temps d'exécution pris, le nombre d'erreurs de cache causées, le nombre de cycles CPU utilisés, le nombre d'instructions exécutées ou encore le nombre de branchements pris par un point de trace. Nous présentons dans cette section les trois solutions les plus susceptibles d'être utilisées dans un contexte temps réel.

ftrace

Le traceur ftrace [46] (*Function Tracer*) est intégré au noyau Linux depuis la version 2.6.27. Son usage premier était de tracer les fonctions appelées à l'intérieur du noyau. Les fonctions peuvent donc être tracées de manière ponctuelle montrant seulement l'appel ou sous forme d'intervalle en instrumentant les entrées et les sorties des fonctions. Cette dernière option permet de récréer un graphe d'appel des fonctions plus simples à la compréhension qu'une liste des appels. Dans sa version actuelle, ftrace supporte aussi des fonctionnalités plus avancées comme le calcul de latence entre le temps de réveil d'une tâche et le début de son exécution [47]. En plus de ces fonctionnalités, il permet de récupérer l'information produite par les points de trace statiques insérés à différent endroit dans le noyau. La configuration de ftrace peut se faire durant l'exécution à travers le système de fichier de ftrace. Les options de configuration comprennent le type de traceur actif, des filtres sur les fonctions tracées, la taille du tampon de trace et l'activation ou la désactivation du traçage.

L'utilisation de ftrace est simple puisqu'il fait partie du noyau Linux. Sa flexibilité est aussi intéressante avec l'usage de filtres et de points de trace insérés dynamiquement dans le noyau. Par contre, comme démontré dans [45], son implémentation engendre un surcoût important pour le traçage de systèmes multicoeurs. Son utilisation est aussi surtout limitée au traçage en mode noyau. Les applications peuvent inscrire des évènements dans le tampon de trace à travers le système de fichier, mais cette opération serait trop lente pour enregistrer un débit élevé d'évènements puisqu'elle nécessite l'utilisation du système de fichier à chaque point de trace rencontré.

LTTng

LTTng est un traceur permettant de récupérer les évènements du noyau Linux et d'applications instrumentées en espace usager [48]. Contrairement à ftrace et perf, LTTng est chargé comme module additionnel au noyau Linux. Le module chargé permet l'activation des points de trace statiques. Le traceur peut aussi enregistrer des points de trace insérés dynamiquement dans le code du noyau (*kprobes*). Du côté du traçage en mode usager, le traceur peut seulement récupérer les points de trace statiques des applications.

Le traceur vise la plus grande performance possible afin d'être utilisable dans une variété de contextes comme les systèmes temps réel avec ressources limitées ou sur des grappes de calcul hautes performances. Pour atteindre cet objectif, un tampon circulaire est alloué à chacun des coeurs du système tracé. Ceci évite la synchronisation entre les points de trace. L'écriture dans le tampon circulaire est faite à partir d'instructions atomiques, de façon à être

réentrante venant d'un contexte d'interruption [49]. La taille de ces tampons est configurable par l'utilisateur. Périodiquement, un démon (*daemon*) consomme les évènements présents dans les tampons circulaires pour les écrire sur un support permanent. Le comportement du traceur lorsque tous les tampons sont pleins est configurable. La première option est d'ignorer les évènements subséquents jusqu'à ce que de l'espace soit rendu disponible par le consommateur. La seconde option est de réécrire les évènements dans le tampon par-dessus les plus anciens. Le premier mode est utile lorsqu'effacer une portion de la trace n'est pas acceptable. Avec ce mode, l'historique est complet jusqu'au début de la perte d'évènements. Ce mode de fonctionnement peut masquer des problèmes intermittents qui surviennent lors de périodes d'activité maximale. Une configuration adéquate de la taille et du nombre de tampons aide à limiter le nombre d'évènements perdus. Le second mode garantit que l'historique menant à une situation problématique est conservé en mémoire. Lorsque la situation est détectée, une capture de l'état des tampons peut être prise afin d'analyser le comportement du système jusqu'à ce point. Puisqu'aucun événement n'est rejeté, il est certain que les tampons contiennent les événements d'intérêt. Par contre, lorsqu'un tampon est plein, une sous-section complète est libérée. Si la configuration des tampons n'est pas optimale, un grand nombre d'événements peut être éliminé.

Le format de trace utilisé par LTTng est un format ouvert nommé le *Common Trace Format* (CTF). La particularité du CTF est qu'il ne se présente pas sous une forme figée et invariable. Chaque trace peut utiliser une représentation binaire différente du moment qu'elle peut être décrite dans par le *Trace Stream Description Language* (TSDL). Ce langage s'apparente au format utilisé pour décrire des structures dans le langage de programmation C. Ainsi, les outils supportant ce format peuvent lire des traces provenant d'une variété de sources.

Perf

Perf est un traceur lui aussi faisant partie du noyau depuis Linux 2.6. Son utilité première est la lecture de compteurs de performance comme le nombre de défauts de cache, le nombre d'instructions exécutées, le nombre de cycles CPU, etc. Des évènements de compteurs logiciels comme le nombre de changements de contexte peuvent aussi être récupérés par perf [50].

En plus des compteurs de performances, Perf permet de profiler les fonctions exécutées en lisant périodiquement le compteur de programme, donnant un aperçu du temps passé dans chacune d'entre elles. Cette approche reste une approximation du résultat qui serait obtenu en instrumentant directement chacune des fonctions. Perf peut aussi enregistrer les points de trace statiques du noyau Linux afin de produire une trace d'exécution [51].

Perf combine une approche statistique par le profilage et la lecture de compteurs de perfor-

mances et une approche déterministe avec les points de trace statiques du noyau. Un désavantage d'une approche par profilage est que des appels de fonctions peu fréquents risquent de ne pas apparaître dans le rapport produit. De plus, tout comme ftrace, perf ne permet le traçage en mode utilisateur.

2.3.2 Traçage de systèmes embarqués

Puisque l'utilisation des systèmes d'exploitation temps réel pour système embarqué est très fragmentée par comparaison au domaine des systèmes d'exploitation classiques, les solutions de traçage sont à évaluer au cas par cas selon le système d'exploitation étudié. La seule façon de fournir un support multiplateforme est de partager une librairie de traçage qui ne fait aucune hypothèse sur la configuration du système sous-jacent (architecture, système d'exploitation, librairie). Ce critère disqualifie toutes les solutions de traçage disponibles sous Linux décrites à la section précédente. Nous proposons dans la suite de cette section une revue des solutions de traçage qui visent à être utilisables sur une multitude de plateformes et systèmes d'exploitation.

Barectf

Barectf [52] est un outil qui permet de générer une librairie de traçage codée en C produisant une trace dans le format CTF. L'outil prend en entrée un fichier au format YAML décrivant le contenu et le format de la trace produite. À partir de ce fichier de configuration, les points de trace pour les différents évènements sont générés et prêts à être utilisés dans une application. Le générateur de code permet de stocker tous les types de base dans le point de trace ainsi que des chaînes de caractères. Par contre, il n'est pas possible d'enregistrer des tableaux ou des structures à partir du code généré, comme le permet la spécification CTF. Pour terminer l'intégration de la librairie de traçage, l'utilisateur de l'outil doit développer lui-même une partie de code spécifique à chaque intégration pour allouer et initialiser les tampons recevant les évènements.

Un avantage évident de barectf est sa flexibilité, puisque l'utilisateur choisit lui-même les points de trace nécessaires qui seront générés. Toutefois, l'ajout de la solution à une application demande un effort d'analyse afin de déterminer les types d'évènements nécessaires. Aussi, l'utilisateur est responsable de la gestion des tampons de trace. Si un tampon circulaire est nécessaire, il devra être ajouté par l'utilisateur. De plus, la librairie de traçage ne génère pas des points de trace réentrants utilisables dans des routines d'interruption, puisque le générateur de code n'a pas connaissance de l'architecture ciblée. L'utilisateur devrait y ajouter son propre code de synchronisation.

Librairie de Tracealyzer

Tracealyzer est une solution de traçage incluant un outil de visualisation et une librairie de traçage. Le code de la librairie de traçage est disponible de façon ouverte pour son intégration avec le système d'exploitation temps réel à code source ouvert FreeRTOS [53], mais sa licence prohibe son utilisation autre que pour les logiciels développés par la compagnie Percepio. La trace est d'ailleurs générée dans un format binaire propriétaire non documentée officiellement.

La librairie implémente une liste de points de trace qui peuvent être insérés aux endroits appropriés dans le code de l'application ou du système d'exploitation étudié [54]. Pour être compatibles avec un système d'exploitation différent, les points de trace nécessitent de légères modifications afin d'utiliser la bonne interface pour récupérer l'identificateur des tâches et des ressources manipulées. Les points de trace sauvegardent l'information selon un des trois modes d'écriture disponibles : sauvegarde instantanée dans un tampon fixe, sauvegarde instantanée dans un tampon circulaire et diffusion en continu vers un circuit ou périphérique spécialisé. L'implémentation du tampon circulaire utilise une taille fixe de bloc d'allocation et un évènement peut s'étendre sur plusieurs blocs si nécessaire. Avant de réécrire un évènement au début du tampon, le traceur doit vérifier que l'évènement écrasé ne s'étend pas sur plusieurs blocs. Ceci est géré en encodant le nombre de blocs utilisés dans l'identificateur unique des évènements concernés. Ces évènements spéciaux ont une plage d'identificateur plutôt qu'un seul identificateur pour éviter les collisions.

La librairie de Tracealyzer fournit une solution quasi clé en main pour tracer les systèmes d'exploitation supportés. Toutefois, l'implémentation est plus difficile à étendre que la solution de barectf.

Grasp

Grasp [55] est une solution de traçage incluant un outil de visualisation basique et une librairie de traçage. Les évènements de la trace sont des commandes Tcl. Ce format est beaucoup moins compact que celui utilisé par barectf ou Tracealyzer, mais simplifie grandement les outils de visualisation. La librairie définit les points de trace utiles pour l'analyse d'ordonnancement du système, puis l'utilisateur les insère aux endroits appropriés selon le cas d'utilisation.

Les fonctionnalités de la librairie de traçage de Grasp sont limitées, le format de trace n'est pas optimal pour une utilisation dans un système avec des ressources restreintes et les points de trace ne sont pas utilisables dans des routines d'interruption. L'outil peut avoir une utilité dans un contexte de prototypage rapide, mais n'est pas assez étendu pour être utile dans un contexte industriel.

2.4 Analyse de traces

Une fois la trace collectée, il est avantageux d'utiliser un outil qui permet de naviguer dans la trace visuellement. Nous proposons ici une liste d'outil qui consomment des traces d'exécution de système temps réel afin d'en extraire de l'information de haut niveau pour aider le développeur d'application.

ConpathView

ConpathView est un outil non disponible publiquement qui permet la visualisation de traces d'exécution de processus ARINC 653 [56, 57]. L'outil est conçu pour assister les développeurs d'application à la recherche d'erreurs causées par l'utilisation concurrente de primitives de synchronisation comme les évènements (*Event*). À partir d'une trace d'exécution, l'outil montre sur un axe vertical les accès en écriture ou lecture à de la mémoire partagée et les appels APEX impliquant des primitives de synchronisation. Lorsque l'ordre des appels de synchronisation risque d'introduire une condition de course, une ligne pointillée relie les deux appels. Lorsque l'ordre des appels de synchronisation est correct, la ligne les reliant est pleine. ConpathView permet de relier la visualisation de la trace d'exécution au code source de l'application. Autre que pour ce cas d'utilisation spécifique, l'utilité de l'outil est assez limitée. Il ne permet pas la visualisation de processus s'exécutant dans des partitions distinctes et n'offre aucune statistique quant au temps d'exécution de l'application.

Les mêmes auteurs proposent d'implémenter un système de monitorage implémentant le même mécanisme de détection d'erreurs de concurrence dans [58]. Comme les erreurs potentielles sont interceptées durant l'exécution, la synchronisation est forcée par le moniteur en insérant une acquisition de verrou dans le chemin du processus fautif. Un désavantage de cette approche est la modification du code durant l'exécution modifiant ainsi le cas de pire temps d'exécution possible.

CoreTAna

CoreTAna est un outil d'analyse de traces de système temps réel permettant de dériver un modèle AUTOSAR valide à partir de la trace d'exécution de l'application pour des fins de rétro-ingénierie [59]. L'outil utilise le format de trace ouvert *Best Trace Format* (BTF). Puisque plusieurs systèmes d'exploitation utilisent un format propriétaire spécifique, les traces doivent d'abord être converties au format BTF. CoreTAna supporte deux niveaux de granularités. Le niveau des processus traite ceux-ci comme des boites noires, ce qui permet de créer un modèle contenant le temps d'activation des processus et leur priorité. Le niveau des fonctions donne

un modèle plus détaillé avec l'arbre d'appel des fonctions et leur temps d'exécution. À partir de ces modèles, CoreTAna peut calculer des métriques utiles aux développeurs d'application telles que le surcoût du système d'exploitation, les patrons d'actions d'activations externes et les temps exacts d'exécution des processus. L'outil s'avère intéressant pour calculer automatiquement un modèle et des métriques réutilisables pour des analyses d'optimisation de l'ordonnancement, mais n'aide pas les développeurs à diagnostiquer des problèmes d'exécution.

Tracealyzer

Tracealyzer est un outil de visualisation de traces de systèmes d'exploitation temps réel, commercialisé par Percepio. Il supporte des traces provenant de plusieurs systèmes d'exploitation dont FreeRTOS, uC-OS, SAFERTOS, ThreadX, ARM Keil RTX5, VxWorks et Linux [60].

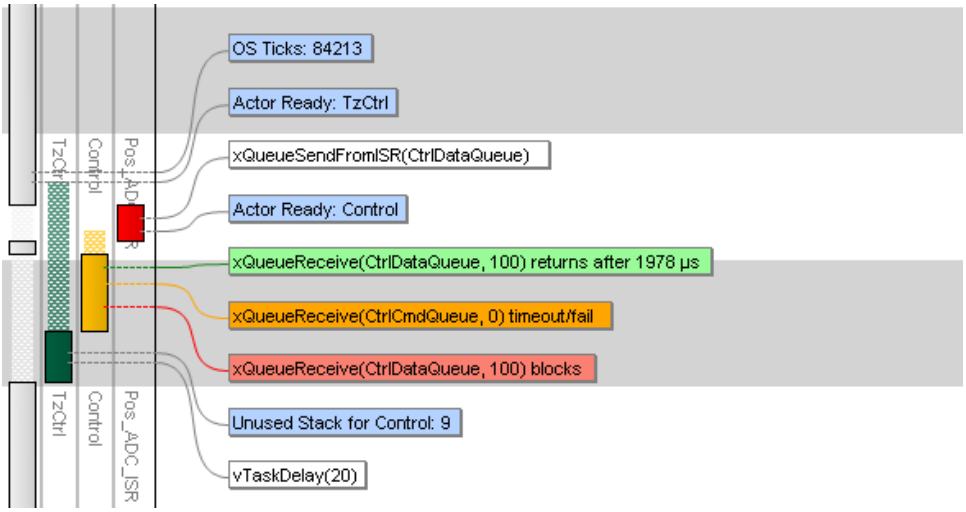


Figure 2.3 Interface de Tracealyzer avec la vue des états des tâches.

La vue principale du logiciel est un graphique vertical en fonction du temps qui montre l'état d'exécution de chacune des tâches présentes dans la trace présentée à la figure 2.3. La trace peut être explorée en se déplaçant dans le temps et en agrandissant les zones d'intérêt. Il est aussi possible de sélectionner une tâche afin d'obtenir de l'information additionnelle à son sujet comme sa priorité, son temps d'exécution et son temps de réponse minimal, moyen et maximum. La vue est par ailleurs augmentée avec des annotations sur les fonctions de l'interface publique du système d'exploitation appelée par les tâches. Les développeurs d'application peuvent aussi se servir de ce système d'annotation en écrivant des messages à travers un service de journalisation fourni par la librairie de traçage.

La librairie de traçage permet aussi aux développeurs d'enregistrer des évènements personnalisés qui n'ont pas d'utilité pour les analyses par défaut. Le logiciel permet de définir des machines à états finis stimulées par ces évènements [61]. Les machines à états finis peuvent être ajoutées à la vue principale du logiciel. Les machines à états peuvent fournir de l'information sur l'état interne de l'application, qui est impossible de connaître pour le système d'exploitation. Le flot de fonctionnements de la machine à état peut être visualisé sous forme d'un graphe pour confirmer qu'elle a été correctement conçue.

La vue principale du logiciel est synchronisée avec les autres vues offertes par le logiciel. Par exemple, la vue de rapport de statistiques montrant le temps d'exécution et le temps de réponse d'une tâche permet de basculer directement au moment dans la trace où le pire temps de réponse a été observé. Une autre vue intéressante basée sur les statistiques est la visualisation dans le temps de la charge utilisée d'un CPU selon les tâches. Un diagramme à bandes empilées montre le détail du temps pris par chacune des tâches avec des couleurs différentes.

Une autre vue très intéressante est celle de graphe de flot de communication. En enregistrant les évènements de création de ressources, comme les mutex, sémaphores et queues de messages, il est possible de créer un graphe de dépendance montrant quelles tâches utilisent quelles ressources. À partir de ce graphe, la liste des évènements reliés à un noeud peut être ouverte.

Trace Compass

Trace Compass est un logiciel de visualisation de traces à source ouverte s'intégrant à l'environnement de développement Eclipse [62]. Une variété de formats de trace peuvent être lus : CTF, GDB, Trace Event ou encore BTF. À partir de l'information contenue dans ces traces, un historique d'état est créé par les différentes analyses disponibles. Cet historique est enregistré par une structure de données sous forme d'arbre nommé le *State History Tree* [63]. Cette structure de données est composée de noeuds chacun contenant un nombre variable d'intervalles représentant des états qui varient dans le temps.

Les vues centrales ouvertes par défaut de Trace Compass sont la vue *Control Flow* et la vue *Resources* montrée dans la figure 2.4. La vue *Control Flow* présente la hiérarchie des processus capturés dans la trace et leurs états d'exécution. La vue *Resource* affiche l'utilisation du processeur par les processus et les routines d'interruptions. Une analyse particulièrement intéressante de Trace Compass et celle du chemin critique [64]. À partir de la vue *Control Flow*, un fil d'exécution peut être sélectionné afin de lancer cette analyse. Le résultat est une nouvelle ligne du temps montrant les dépendances et les états de tous les fils d'exécution ayant

réveillé le fil d'exécution sélectionné. Cette analyse permet, entre autres, de trouver les causes de mise en attente de fils d'exécution résidant sur des machines distantes communiquant par le réseau.

Trace Compass peut être étendu de façon dynamique à l'aide d'analyse basée sur le langage de représentation de données XML [65]. L'utilisateur peut décrire une machine à état fini stimulée par les évènements de la trace et définir des actions à prendre lors des transitions. Ces actions permettent de construire un historique d'état différent de celui créé par les analyses de base. L'historique d'états peut ensuite être montré à l'utilisateur par une vue générique.

Les analyses accessibles dans Trace Compass ne visent pas spécifiquement les systèmes temps réel. Par contre, puisqu'il s'agit d'un logiciel à code source ouvert facilement extensible, il peut être adapté pour fournir des analyses ciblées pour les systèmes temps réel. Par exemple, dans [66] les auteurs proposent d'étendre Trace Compass afin de miner automatiquement les patrons d'exécution de tâches périodiques et d'identifier les occurences ayant un temps d'exécution nettement supérieur à la moyenne.

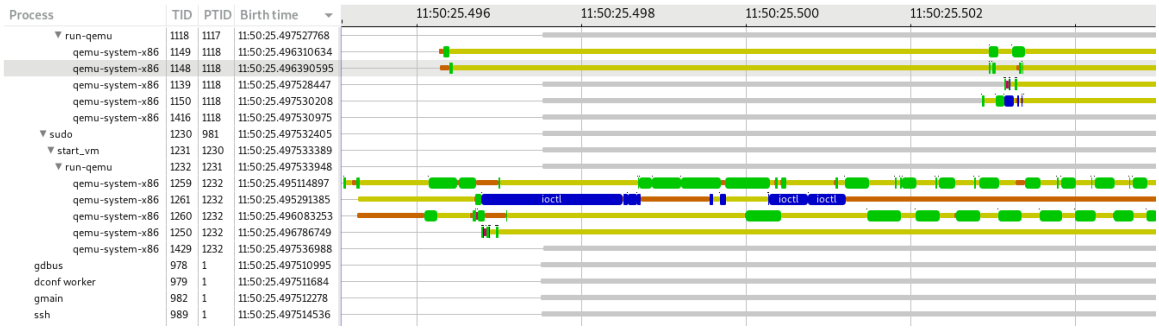


Figure 2.4 Interface de Trace Compass avec la vue *Control Flow*.

QNX System Profiler

QNX System Profiler [67] est un outil d'analyse de traces développé pour visualiser et explorer les traces provenant du système d'exploitation temps réel QNX. L'outil offre trois vues principales. La première montre le taux d'utilisation des processeurs. Ce taux est séparé entre le temps consommé par l'utilisateur et le temps consommé par le noyau. La seconde vue montre l'état des tâches du système dans le temps dans un diagramme de Gantt. Cette vue peut être filtrée selon les fils d'exécution. De plus, en sélectionnant un sous-groupe de fils d'exécution, il est possible de basculer à la vue d'utilisation des processeurs, montrant maintenant l'utilisation des processeurs selon le groupe de fils d'exécution choisi. La troisième vue permet de calculer des statistiques de temps d'exécution dans des applications qui

suivent un mode de communication client-serveur détecté automatiquement. Cette analyse unique à QNX System Profiler donne le temps d'exécution du client et le temps imposé par le client au serveur, suite à une requête. Ces métriques sont utiles pour identifier le client le plus gourmand qui bénéficierait le plus d'un effort d'optimisation. Cette vue se rapproche de l'analyse de chemins critiques offerte par Trace Compass, mais ne permet pas d'observer l'état de toutes les tâches qui causent ces temps d'attente.

RapiTask

RapiTask [68] est un outil de visualisation commercialisé par Rapita Systems. Il s'intègre à RapiTime, un second outil développé par la compagnie, qui permet d'évaluer le pire temps d'exécution d'une tâche. RapiTask permet d'explorer une trace d'exécution dans un diagramme de Gantt selon le temps, montrant l'état des différents fils d'exécution sur le système. Les données collectées par RapiTime permettent de faire des recherches de patrons d'exécution dans la trace définis selon la périodicité des tâches, leurs temps d'exécution et leurs temps de réponse. Ce système de filtre est très avantageux pour effectuer des recherches dans des traces de grandes tailles. Par contre, l'outil ne permet pas la visualisation à travers les partitions.

Wind River Workbench

Wind River Workbench est un environnement de développement intégré développé spécifiquement pour les besoins du système d'exploitation VxWorks. Il contient un outil de visualisation de traces intitulé *System Viewer*. L'outil permet la visualisation de traces provenant de Linux ou VxWorks [69].

La visualisation principale de l'outil est une ligne du temps montrant l'état des processus, le processeur sur lequel ils sont exécutés, les interruptions et l'utilisation de ressources allouées par le noyau comme les sémaphores ou les mutex. Pour savoir sur quel processeur s'exécute un processus, il faut repérer l'annotation au début de son temps d'exécution. L'utilisation de ressources allouées par le noyau est identifée par différents symboles superposés à la ligne d'exécution. L'outil permet aussi une analyse détaillée de l'utilisation de la mémoire allouée aux applications. La liste de toutes les allocations ainsi que leurs tailles est affichée à l'utilisateur. En plus d'analyser l'information issue de traces, cet outil fonctionne avec l'information obtenue d'un profileur. Le profileur lit périodiquement le compteur de programme de façon à reconstruire une image à haut niveau du temps passé dans chacune des fonctions.

Wind River Workbench est une solution complète offrant beaucoup d'analyses aux dévelop-

peurs. Toutefois, son usage est limité à Linux ou VxWorks. De plus, la visualisation de traces multicoeur est très chargée, de sorte qu'il est difficile de voir sur quel processeur un processus s'exécute. Une solution préférable serait de répartir l'information sur des vues synchronisées, l'une montrant l'état des processus et la seconde montrant l'état des processeurs comme le fait Trace Compass.

2.5 Banc d'essai

Les bancs d'essai de système d'exploitation temps réel permettent de caractériser et de comparer leurs performances. Nous présentons dans cette section une revue des bancs d'essai développés avec cet objectif en tête. Les bancs d'essai peuvent être groupés en deux catégories distinctes selon la méthode choisie pour leur conception [70, 71].

Synthétique Un banc d'essai synthétique n'accomplit aucune tâche particulière et est conçu spécifiquement pour les besoins du test.

Applicatif Un banc d'essai applicatif est dérivé d'une application réelle et modifié pour des fins de tests.

Nous notons aussi une séparation selon la façon dont les résultats sont collectés par le banc d'essai [72].

Granularité fine le banc d'essai permet d'obtenir des mesures prises dans un contexte précis pour un service du système d'exploitation.

Orienté applicatif le banc d'essai fournit une vue plus globale des performances du système d'exploitation dans un contexte applicatif.

Rhealstone

Rhealstone est un banc d'essai synthétique à granularité fine proposé par Kar et Rabindra et implémenté par Kar pour le système d'exploitation iRMX [73]. Le banc d'essai comporte six scénarios de test désignés comme étant les cas les plus importants afin de mesurer les performances d'un système d'exploitation temps réel.

Changement de contexte (1) Deux tâches de même priorité sont démarrées et se passent la main en continu grâce au service du système d'exploitation qui permet un ordonnancement coopératif.

Changement de contexte (2) Une tâche de haute priorité et une tâche de basse priorité sont démarrées. La tâche de basse priorité fait du travail en continu, puis est préemptée périodiquement par la tâche de haute priorité.

Latence d'interruption Une tâche génère volontairement une interruption matérielle qui est récupérée par une routine d'interruption.

Manipulation de sémaphore Deux tâches de même priorité sont instanciées. Le corps des tâches est le même : un sémaphore partagé est acquis puis libéré en boucle.

Manipulation de mutex Trois tâches de priorités croissantes sont créées. Les tâches de faible priorité et de haute priorité tentent d'acquérir le même mutex. La tâche de priorité moyenne ne contient qu'une boucle de travail. La tâche de basse priorité s'exécute en premier grâce à un délai inséré dans la tâche de haute priorité, ce qui cause une situation d'inversion de priorité que le système d'exploitation doit gérer.

Échange de messages Deux tâches de priorité différentes s'échangeant des messages sont créées. La tâche de haute priorité reçoit les messages alors que la tâche de basse priorité les envoie.

Dans tous les scénarios autres que la mesure de la latence d'interruption, la méthodologie pour la prise de mesure est la même. Le scénario est exécuté une première fois sans utiliser des services du système d'exploitation décrit dans le test, puis une seconde fois en utilisant les services du système d'exploitation. La différence entre ces deux temps est le temps pris par le système d'exploitation pour accomplir la tâche demandée. La latence d'interruption est mesurée de façon plus précise en prenant le temps avant le déclenchement de l'interruption matérielle et au début de la routine d'interruption.

Un banc d'essai dérivé de Rhealstone est implémenté dans [74]. Les auteurs emploient toutefois une méthode de mesure plus précise. Deux signaux sont envoyés à un oscilloscope à travers le connecteur GPIO, un au départ de la zone d'intérêt et un second à la fin. Cette approche est reprise dans [75] pour la comparaison des performances entre CMSIS-RTOS et X-Real Time Kernel de eSystech. Les auteurs de [76] proposent une méthode alternative en exécutant le banc d'essai sur un système sur puce composé d'un processeur et d'un circuit logique programmable (FPGA). Le circuit logique programmable est reprogrammé pour implémenter un minuteur matériel ainsi qu'un circuit lisant des signaux envoyés par les tâches pour lesquelles le temps de changement de contexte est mesuré. Ces raffinements de la méthode de mesure permettent d'obtenir des résultats précis sans que le banc d'essai n'ait de dépendance sur le matériel, mais oblige l'utilisateur à avoir accès à de l'équipement spécialisé.

L'approche générale de Rhealstone est critiquée dans [77] puisqu'elle ne mesure pas le pire

temps d'exécution observé lors des scénarios de test. De plus, les scénarios sont très basiques, ne contenant toujours que deux ou trois tâches, sans possibilité de configuration.

Dhrystone pour RTOS

L'auteur de [78] propose d'utiliser le banc d'essai Dhrystone mesurant la vitesse d'exécution du système comme bloc de base pour construire un banc d'essai synthétique mesurant les performances du système d'exploitation. La performance des services mesurée par les scénarios constituant le banc d'essai est similaire à ceux proposés par Rhealstone. L'approche pour la création de ceux-ci est cependant différente.

Changement de contexte (1) Une à vingt tâches de même priorité sont créées. Les tâches sont interrompues de façon périodique afin de passer le contrôle à la suivante. La période d'interruption est configurable.

Changement de contexte (2) Une à vingt tâches de priorité croissante sont créées et démarrées. Une interruption périodique suspend la tâche de plus haute priorité restante. Lorsqu'il ne reste que la tâche de plus basse priorité, l'interruption change de mode pour réveiller les tâches de plus haute priorité.

Manipulation de sémaphores Deux tâches de priorité différentes sont créées. La tâche de basse priorité s'exécute puis acquiert et libère un sémaphore en continu. La tâche de haute priorité est réveillée par une interruption puis tente d'acquérir puis de libérer le même sémaphore.

Latence d'interruption Une tâche de haute priorité est mise en attente d'un signal émis par une routine d'interruption. Lorsque la tâche s'exécute, elle émet un signal à un oscilloscope.

Échange de messages Dix tâches sont démarrées. Neuf d'entre elles attendent un message de leur voisin et le transmettent à la suivante. La dernière tâche envoie un ou deux messages à la première tâche de la boucle selon le mode de configuration actuelle. Le mode de configuration est changé périodiquement par une routine d'interruption.

Allocation de mémoire Une à vingt tâches de même priorité sont créées. Une interruption périodique suspend la tâche courante et réveille une nouvelle tâche. Chaque tâche alloue ou désalloue de la mémoire.

Chacune des tâches de travail décrite dans les scénarios ci-dessus exécute le corps de la boucle Dhrystone. Le nombre total d'exécution de cette boucle à travers toutes les tâches est comptabilisé. Le surcoût du système d'exploitation est estimé en comparant ce score à

une mesure de base obtenue en exécutant une seule tâche sur le système qui exécute cette même boucle. Le banc d'essai permet d'obtenir une idée globale des performances du système d'exploitation pour des fins de comparaison, mais ne donne aucune indication sur les pires cas d'exécution ni de résultats concrets pouvant être utilisables pour une analyse de faisabilité d'ordonnancement.

Hartstone

Hartstone est une spécification de banc d'essai synthétique initialement développée pour le langage de programmation Ada [79]. Ce banc d'essai vise à évaluer les capacités temps réel d'un système d'exploitation. Les résultats de ce banc d'essai ne sont pas exprimés en termes de surcoût. Ils indiquent plutôt si le système conserve ses propriétés temps réel dans cinq scénarios prédéfinis.

Tâches périodiques avec fréquences harmoniques (TH) Un nombre configurable de tâches périodiques sont instanciées avec des fréquences étant un multiple de la plus petite fréquence.

Tâches périodiques avec fréquences non harmoniques Un nombre configurable de tâches périodiques sont instanciées avec des périodes sans restrictions

TH et tâches apériodiques En plus d'un certain nombre de tâches périodiques avec des fréquences harmoniques, un certain nombre de tâches réagissant à des interruptions externes sont créées.

TH et synchronisation Un nombre configurable de tâches périodiques avec des fréquences harmoniques utilisant des primitives de synchronisations entre elles sont instanciées.

TH avec synchronisation et tâches apériodiques Ce scénario combine tous les types de tâches définis ci-dessus.

Les spécifications des scénarios sont laissées délibérément vagues. Ainsi, l'implémentation exacte du banc d'essai peut être modelée de façon à représenter le mieux possible le cas d'utilisation du système d'exploitation testé. Le nombre de tâches et les fréquences sont configurables, ce qui permet de trouver le point de défaillance du système par rapport aux échéances des tâches.

Hartstone est étendu dans [80] comme banc d'essai pour des systèmes distribués avec cinq nouveaux scénarios permettant de tester les capacités de communication temps réel entre un client et un serveur. Les scénarios décrits dans la spécification de Hartstone sont utilisés dans [81] pour comparer l'efficacité d'algorithme d'ordonnancement de tâches temps réel.

RT-Test

RT-Test est un banc d'essai synthétique ciblant Linux [82]. L'objectif du banc d'essai est de mesurer les latences maximales, minimales et moyennes du système sous ses conditions d'opération normales. Par exemple, le scénario intitulé *cyclictest* permet de vérifier la latence d'ordonnancement d'une tâche périodique en calculant la différence de temps entre le temps prévu de l'exécution de la tâche et son exécution actuelle [83].

Les tests évaluant la latence des sémaphores, des mutex, de l'échange des messages et de l'échange de signaux ont tous la même structure que *cyclictest* (*svsematest*, *ptsematest*, *pmqtest*, *signaltest*). Deux tâches sont instanciées, la première signalant la primitive de synchronisation et la seconde se mettant en attente de celle-ci. Le temps entre la relâche de la tâche mise en attente et le début de l'exécution de celle-ci est un indicateur de la latence du système.

RT-Test contient aussi des scénarios spécifiques au noyau Linux. Le test *backfire* évalue le temps pris pour envoyer un signal d'un pilote à une application en mode usager. Le test *hackbench* crée un nombre configurable de paires de fils d'exécution s'échangeant des messages à travers des *socket* UNIX afin d'évaluer les performances de l'ordonnanceur. Le test *rt-migrate-test* crée un groupe de tâches temps réel pour s'assurer qu'elles s'exécutent en priorité sur les différents coeurs disponibles. Le test *pi_test* crée des scénarios de priorité d'inversion afin de vérifier qu'ils sont gérés correctement.

Thread-Metric

Thread-Metric est un banc d'essai synthétique proposé par la compagnie Express Logic [84]. Le banc d'essai est composé de sept scénarios, similaires à ceux proposés par Rhealstone.

Changement de contexte (1) Cinq tâches de même priorité sont créées. Chacune passe la main à la tâche suivante.

Changement de contexte (2) Cinq tâches de priorités croissantes sont instanciées. La tâche de plus basse priorité est démarrée et les autres sont laissées en attente. La tâche s'exécutant réveille la tâche suivante de priorité supérieure. Lorsque toutes les tâches sont réveillées, elles se suspendent une à une.

Gestion d'interruption sans préemption Une tâche génère une interruption matérielle. La routine d'interruption est exécutée, puis le contrôle revient à la tâche initiale.

Gestion d'interruption avec préemption Une tâche de basse priorité génère une interruption matérielle. La routine d'interruption exécutée réveille une tâche de haute priorité.

Manipulation de sémaphore Une tâche signale un sémaphore puis attend le même sémaphore en boucle.

Échange de message Une tâche envoie et lit en boucle un message dans une queue de message.

Allocation de mémoire Une tâche alloue un bloc de mémoire puis le désalloue en boucle

Dans chacun des scénarios de test décrits, les tâches instanciées incrémentent un compteur à chaque tour de boucle. La moyenne du compteur de chacune des tâches donne le résultat pour le scénario. Cette façon de mesurer le surcoût ignore les pires cas d'exécution, mais permet une comparaison rapide sans l'utilisation d'un minuteur précis. Par exemple, dans [85] le scénario de test de changement de contexte coopératif est utilisée afin de comparer la disponibilité du système selon le débit des communications dans un contexte d'Internet des objets. Toutefois, le banc d'essai ne permet pas d'obtenir des résultats précis quant aux performances du système d'exploitation. De plus, contrairement à Rhealstone, les scénarios de manipulation de sémaphore et d'échange de message calculent le temps d'utilisation du service lorsqu'aucun changement de contexte n'est initié.

2.6 Conclusion de la revue de littérature

Notre revue de littérature démontre que malgré l'existence d'outils pour l'analyse de traces d'exécution de systèmes temps réel, aucun ne semble actuellement conçu spécialement pour le partitionnement spatial et temporel. Pourtant, nous dénotons l'existence de problèmes particulièrement difficiles à diagnostiquer dans un contexte de programmation parallèle sans l'existence de tels outils. Principalement, nous identifions des lacunes dans la capacité des outils à présenter clairement l'ordonnancement des partitions lors de l'exécution et à souligner la présence d'interférence dans les systèmes multicoeurs.

De plus, nous remarquons qu'il n'existe aucun banc d'essai portable et ouvert permettant d'obtenir des résultats clairs sur la performance de systèmes d'exploitation temps réel avec partitionnement spatial et temporel. Les bancs d'essai disponibles en ce moment sont soit basés sur une approche de mesure comparative peu précise ou soit très simple. Il y a donc un intérêt à fournir un nouveau banc d'essai comblant ces lacunes.

CHAPITRE 3 MÉTHODOLOGIE

Ce chapitre présente la méthodologie employée pour l'accomplissement du projet de recherche. Nous faisons d'abord une brève description des tâches à réaliser afin d'accomplir les objectifs mis de l'avant à la section 1.3, puis nous détaillons l'environnement de travail, les outils utilisés ainsi que les travaux complétés lors du projet.

3.1 Tâches à réaliser

L'objectif principal de la recherche est de développer un environnement d'analyse de traces pour les systèmes d'exploitation temps réel avec partitionnement spatial et temporel. La première étape de la recherche est donc d'identifier des systèmes d'exploitation à code source ouvert pouvant servir de support à notre recherche. Dans la revue de littérature, quelques projets de ce genre ont été présentés. Une fois ce choix fait, la seconde étape est d'établir une méthode de collection de traces qui permet de récupérer l'information tant sur l'ordonnancement des partitions du système que des tâches à l'intérieur des partitions. Cette étape est cruciale pour la réussite du projet puisque le développement des outils d'analyse de traces ne peut se faire sans traces de systèmes d'exploitation. Une fois ceci accompli, la troisième étape est de déterminer les éléments manquants dans les outils de visualisation qui rendent leur utilisation inadéquate pour le type de systèmes étudiés et d'implémenter une vue comblant ces manques dans le logiciel de visualisation de traces Trace Compass. Cette vue devra permettre à l'utilisateur d'explorer rapidement le contenu d'une trace afin d'y identifier des cas d'exécution problématiques. Afin de valider l'efficacité et l'utilité des outils d'analyses de traces développés, la quatrième étape sera de développer un banc d'essai servant à la caractérisation de performances de systèmes temps réel avec partitionnement spatial et temporel. Comme le partitionnement n'est qu'une fonctionnalité supplémentaire d'un système d'exploitation, ce banc d'essai se doit d'être portable sur n'importe quel système temps réel. Finalement, avec tous ces éléments en main, la dernière étape sera de valider l'approche de visualisation proposée en réalisant une étude comparative des performances des systèmes d'exploitation temps réel avec partitionnement spatial et temporel sélectionné.

3.2 Environnement de travail

Deux expériences sont réalisées, la première pour montrer la portabilité du banc d'essai construit et la seconde pour montrer l'utilité de l'environnement d'analyse développé. Chaque

expérience se déroule sur un environnement de travail différent présenté dans les sous-sections suivantes.

3.2.1 Matériel

Le premier environnement de travail est composé d'un système sur puce de type Raspberry PI 2B+ basé sur un processeur multicoeur Cortex A7 d'architecture ARM. Ce système est choisi puisque l'architecture ARM est souvent supportée par les systèmes d'exploitation temps réel, ce qui rend leurs utilisations plus simples pour les besoins de l'étude. Le second environnement de travail est un ordinateur fixe plus traditionnel basé sur l'architecture X86. Les configurations matérielles de ces deux environnements sont présentées dans le Tableau 3.1. Le processeur utilisé sur la station de travail est composé de huits coeurs logiques répartis sur quatres coeurs physiques identifiés comme les CPU 0 à 3 dans la figure 3.1.

3.2.2 Logiciel

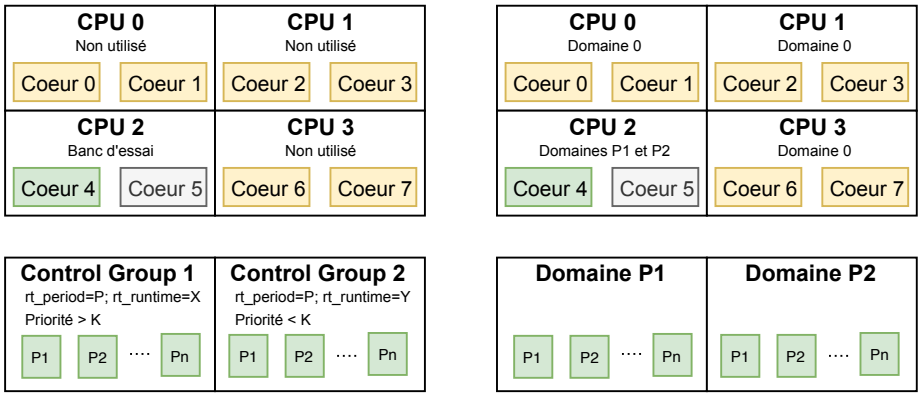
Pour le système sur puce du premier environnement de travail, trois systèmes d'exploitation temps réel à code source ouvert sont utilisés dans l'expérience : FreeRTOS v10.1.1, RTEMS v4.11.3 et Linux v4.14. Lors de l'exécution des tests, la fréquence du processeur est constante et configurée à 900MHz à l'aide du fichier de configuration de démarrage du Raspberry Pi. Puisque FreeRTOS et RTEMS sont des systèmes d'exploitation temps réel beaucoup plus minimalistes que Linux, nous avons manuellement vérifié que leur code de démarrage active les caches de données ainsi que l'unité de gestion de la mémoire.

Le second environnement de travail est utilisé avec Linux v4.19.3 et l'hyperviseur Xen v.4.13. Les coeurs sur lesquels les tests sont exécutés sont isolés afin qu'aucune tâche imprévue

Tableau 3.1 Configurations matérielles des environnements de travail

Élément	Quantité	Type
Raspberry Pi 2B+		
Système sur puce	1	Broadcom BCM2836
Processeur	1	Cortex A7 avec quatre coeurs cadencés à 900 MHz
Mémoire vive	1	1GB
Station de travail		
Carte mère	1	ASUSTeK VANGUARD B85
Processeur	1	Intel Core i7-4790, avec huit coeurs cadencés à 3,6GHz
Mémoire vive	4	Kingston DDR3 1600MHz de 8GB

n'interfère avec les résultats. Ce second environnement permet l'étude du comportement de systèmes d'exploitation avec partitionnement spatial et temporel en suivant les configurations illustrées dans les Figures 3.1 (a) et (b). Pour Linux, chaque partition est représentée par un groupe de contrôle ayant une limite de temps d'exécution fixe. Pour s'assurer que l'ordre entre plusieurs partitions est respecté, les priorités des tâches du premier groupe de contrôle sont strictement supérieures à celles des autres partitions. En suivant cette méthode de configuration, il n'est pas possible d'exécuter plus d'une fois une partition par période d'exécution. Pour Xen, la configuration est plus simple comme l'ordonnancement des domaines selon un politique ARINC 653 est disponible par défaut.



(a) Configuration de Linux (b) Configuration de Xen

Figure 3.1 Configuration des systèmes pour le partitionnement spatial et temporel

3.3 Outils

Les outils pour la visualisation de trace ainsi que la collection de traces ont été présentés dans la revue de littérature. Nous présentons maintenant les outils sélectionnés pour les besoins de l'étude.

3.3.1 Trace Compass

Trace Compass est un logiciel de visualisation de traces à code source ouvert très bien adapté pour les traces du noyau Linux. Étant basé sur Eclipse, le logiciel est facilement extensible à travers un système de plugiciels. Puisque nous étudierons le partitionnement spatial et temporel avec le noyau Linux et des domaines de Xen exécutant Linux, l'utilisation de ce logiciel est tout indiquée. Toutes les vues et analyses proposées sont implémentées comme module d'extension à celui-ci. Le travail se base sur la version 4.3 de Trace Compass.

3.3.2 LTTng

Comme nous l'avons mentionné dans la revue de littérature, LTTng est le traceur du noyau Linux présentant le meilleur déterminisme dans son surcoût. Puisque nous nous intéressons aux systèmes temps réel, l'utilisation de ce traceur est naturelle. LTTng produit des traces dans le format CTF qui est déjà supporté par Trace Compass. De plus, le traceur permet d'enregistrer des évènements provenant d'applications s'exécutant en mode usager, ce qui nous a été utile afin de capturer l'état du système de fichier des groupes de contrôles sur Linux et pour synchroniser les traces entre les domaines et l'hyperviseur Xen. La version 2.10 de LTTng est utilisée pour la création des traces dans les expériences.

3.3.3 Xentrace

Xentrace est un outil qui lit la trace produite par l'hyperviseur Xen afin de l'enregistrer sur le disque. Cet outil est utilisé afin de récupérer l'information sur l'ordonnancement des domaines lors des expérimentations. Xentrace ne produit pas de traces dans le format CTF, mais la sortie peut être convertie dans un format textuel grâce à un script nommé Xentrace_format. Afin d'éviter de devoir créer un module d'analyse syntaxique dans Trace Compass pour le format de trace binaire utilisé par Xentrace, nous avons développé un script qui convertit le format textuel de Xentrace au format CTF.

3.4 Travail réalisé

En ayant suivi les étapes définies plus tôt, deux principaux artéfacts sont produits. Le premier est le banc d'essai pour la caractérisation de performances de systèmes temps réel avec partitionnement spatial et temporel. Le second est une vue pour le logiciel Trace Compass adaptée pour Linux et Xen proposant une technique de visualisation qui pourrait être reprise sur d'autres systèmes temps réel. Le banc d'essai, présenté en détail dans l'article du chapitre 4, est développé avec le langage C. Le banc d'essai est conçu de façon à être facilement portable vers un nouveau système d'exploitation. Pour atteindre cet objectif, une interface interne est utilisée plutôt que d'appeler directement les services du système d'exploitation testé. La vue dans Trace Compass, elle aussi discuté dans l'article du chapitre 4, permet de visualiser l'ordonnancement des partitions ainsi que des tâches du système tracé. Nous avons aussi intégré l'environnement d'analyse de traces au banc d'essai proposé afin d'en faciliter l'analyse des résultats.

CHAPITRE 4 ARTICLE 1 : BENCHMARKING AND TRACING OF SPACE AND TIME PARTITIONING REAL-TIME OPERATING SYSTEMS

Authors

Guillaume Champagne <guillaume.champagne@polymtl.ca>
Michel Dagenais <michel.dagenais@polymtl.ca>

Keywords : Space and time partitioning, Benchmark, Tracing, Trace analysis

Submitted to : Real-Time Systems

Abstract

The current trend in commercial avionics is to share resources among multiple real-time applications to reduce cost and mass. To maintain the performance of these applications, the use of multicore processors becomes very tempting. The ARINC 653 standard specifies the interfaces and behaviour of operating systems with space and time partitioning. This provides robust isolation guarantees to enable the use of parallelism in safety-critical real-time applications. Characterizing the performance of these operating systems and understanding the root cause of the performances issues can be a daunting task without the appropriate tools. To tackle this issue, we propose a novel open-source benchmark, especially created for these systems, which either produces highly accurate results on its own or can be integrated with a trace analysis framework to provide in-depth details into the results. The benchmark measures the performance of common operating systems services as well as the jitter of the partitions. The trace analysis tool includes a view to show the scheduling and the states of the ARINC 653 partitions and the processes they contain. This tool was used to investigate performance issues detected while preparing comparative performance analyses for two sets of operating systems.

4.1 Introduction

Real-time systems are designed to accomplish a very narrow set of functions. They interact with their environment by receiving input from it and producing output sent to other computer or electronic systems. While their use is limited to their application domain, their complexity and number rise continuously. On systems integrating a great number of software components, monetary cost, energetic demand and mass increase quickly if each is treated as a black box with its own resources. In commercial avionics, this issue was identified early on and tackled with the use of integrated modular avionics.

Integrated modular avionics is a way of organizing and conceiving systems for planes in which resource sharing is put forward. However, by their very critical nature, these systems cannot use traditional operating systems to share resources among software applications. The ARINC 653 standard was published to regulate the development of a new generation of real-time operating systems: operating systems with strict space and time partitioning. The goal of these operating systems is to provide robust partitioning between software applications sharing hardware resources. By providing strong guarantees, such an operating system allows the cohabitation of applications with different design assurance levels.

The ARINC 653 standard proposes to implement space and time partitioning with the use of software partitions containing processes. Processes within a partition share the same address space, but partitions do not. Temporal partitioning is enforced by the use of a static periodic schedule in which time slots are reserved for each partition. Consequently, processes from one partition cannot corrupt memory owned by another partition or steal time from other applications.

When very complex applications share resources, it becomes highly appealing to use multicore processors to enable parallelism at every level of the system. Ideally, partitions could run their multitask applications in parallel while providing the same robust partitioning. To fully understand the performance of these systems in parallel contexts, advanced analysis tools are required. To tackle this problem, we propose the use of kernel tracing and trace visualization. To demonstrate the validity of the solution, we show how tracing can help to understand the results of RTOSBench [1], an open source portable benchmark that we developed especially for real-time operating systems with space and time partitioning.

The rest of the paper is structured as follows. Section 4.2 discusses the related work in the domain of real-time operating systems tracing and benchmarking. Section 4.3 presents how the benchmark can be ported to any operating system. In section 4.4, we detail the content

1. `github.com/gchamp20/RTOSBench`

of the proposed benchmark. Section 4.5 demonstrates how tracing can be used to analyze the results of the benchmark. In section 4.6 and 4.7, we present the results of the performance analysis of traditional real-time operating systems and of real-time operating systems with space and time partitioning, based on our benchmarking method. We conclude in sections 4.8 and 4.9 with a discussion on the limitations and future improvements to the trace analysis method proposed.

4.2 Related work

In this section, we review previous studies that proposed benchmarks for performance analysis of real-time operating systems and discuss the state of the art in tracing for real-time operating systems.

A few benchmarks to evaluate the performance of real-time operating systems have been implemented and distributed, like Rhealstone [73], the benchmark of Mcrae [78], Thread-Metric [84] and Harstone [79]. These benchmarks propose scenarios in which metrics about context switches, handling of synchronization primitives, communication mechanisms and interrupts are gathered. In the benchmark of McRae and Thread-Metric, a comparative approach, based on the number of iterations of a test achieved in a fixed time period, is preferred over time measurements. This makes these benchmarks only useful for direct comparison of operating systems, since timing measurements are not produced. Rhealstone takes time measurements and proposes to average the results over all the scenarios to get a single number used for comparison, like the well-known Dhrystone [86] benchmark for processor execution speed. The results of Rhealstone may be presented separately, but the test scenarios remain very simple and never involve more than three tasks. The Harstone benchmark specifies generic test scenarios in which tasks have configurable deadlines. The deadlines are tightened until failures are observed. The point of failure can be compared between operating systems, but the results fail to give the user insight into the overhead to be expected while using the operating system. None of the benchmarks proposed until this point cover operating systems with space and time partitioning. Han and Jin compared the performance of three partitioning designs on Linux, but their benchmarking method is not standardized and is thus hard to reproduce [87].

Tracing has been shown to be efficient for real-time systems. Beamonte et al. studied the latency of the LTTng kernel and userspace tracer to prove that it is sufficiently deterministic to be used to troubleshoot real-time performance problems [83]. The LTTng tracer has since been used in other studies to analyze the performance of real-time systems. Côté et al. proposed a method to automatically detect execution patterns in real-time Linux kernel

traces and to compare the execution time of the patterns occurrences [66]. Desfossez et al. proposed a run-time latency tracker which can be used to record a trace snapshot when a higher than expected latency occurs [88]. There is, however, a lack of research on how the same principles could be applied to operating systems with space and time partitioning.

Collecting trace data is a first step in understanding the behavior of the studied system. However, looking through the content of a trace is often easier with the use of a visualization tool. A number of tools provide a trace visualization framework for real-time systems, like RapiTime [68], TraceAlyzer [60] or Trace Compass [89]. These tools propose a timeline view in which all the recorded states of the tasks are displayed. However, none of them offer specialized views for real-time operating systems with space and time partitioning. WindRiver Workbench [69] offers a view listing the tasks per partition, but this tool is only compatible with the proprietary operating system VxWorks. In the open source community, the closest we could find in terms of partitioning is the work to trace virtual CPU state through host tracing by Nemati et al. [1]. However, this work does not consider real-time issues in the scheduling of virtual machines or the temporal isolation between them.

4.3 Benchmark structure

To make the implementation of the benchmark portable across a variety of operating systems, it cannot rely on a particular programming interface being present or on data types being consistent. Therefore, the test scenarios of our implementation never directly call the services of the operating system or manipulate its data types. Rather, a porting layer conforming to an internal interface encapsulates the interactions with the tested operating system. The test scenarios only reference this internal interface.

The porting layer consists of three files: an interface header file, a configuration header file and a porting layer implementation file. The interface header file declares the functions that have to be implemented by the porting layer to make the benchmark compatible with an operating system. This file never changes since all the types it uses are either basic types (integer, char, etc.) or type aliases. The underlying types of the type aliases are set in the configuration header file using C *typedef*. This abstraction allows us to write test cases that never directly refer to the types defined by the operating system. The third file, the porting layer implementation file, provides the definition of the functions declared in the interface header file. There are 28 functions to implement to make all the test scenarios usable. However, most of the test scenarios only require a subset of those functions. A list of the required functions accompanies each test scenario to enable partial implementations. The diagram in Figure 4.1 summarizes the relationships between the files forming the porting

layer.

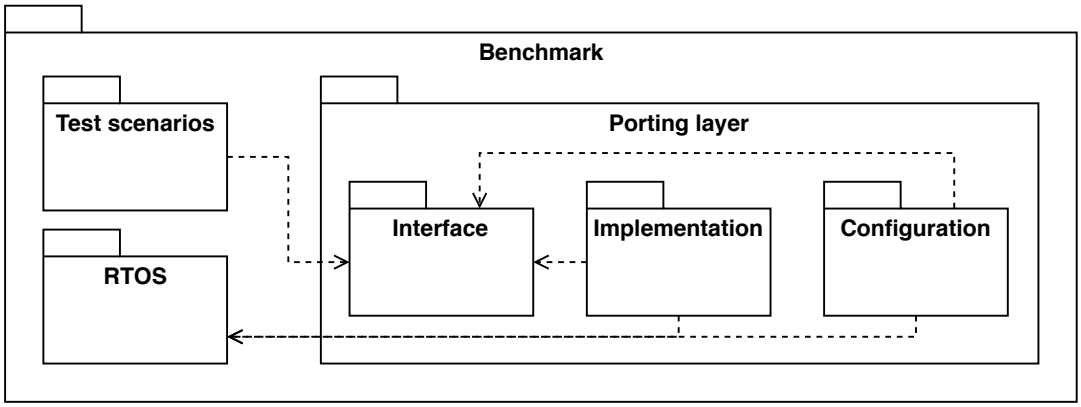


Figure 4.1 Diagram of the benchmark structure

The dependencies in the benchmark structure are either towards the porting layer interface or towards the real-time operating system. Since the test scenarios only depend on the interface, we can claim that they are truly independent of the operating system. The implementation and configuration files both depend on the interface and on the real-time operating system. They depend on the interface, since they provide the definition of the functions and the types aliases used by the test scenarios. If the interface changes, those two files may require modifications. They depend on the operating system, since they access its programming interface and they manipulate the data types it defines. This dependency is acceptable since it is confined to the porting layer portion of the benchmark.

4.4 Test scenarios

The test scenarios part of the benchmark were crafted in a way that makes it possible to record many samples of the same metric. These metrics are presented in the following sections. For each test scenario, a figure shows the expected execution and the metric measured. The metrics are displayed in the figures as horizontal arrows connecting two dotted vertical lines. In the reference implementation of the benchmark, each of these intervals is recorded and used to update the statistics of the test (mean, maximum and minimum value), or kept in memory and printed on the standard output for offline analysis at the end of the test.

The proposed scenarios exercise the core services available in the vast majority of real-time operating systems. We propose a first category of scenarios covering interrupts, cooperative scheduling, semaphores, mutexes and message queues. On top of this, a second category of scenarios is proposed to assess the robustness of temporal partitioning in operating systems

with space and time partitioning. To our knowledge, this is the only publicly available benchmark proposing this approach.

4.4.1 Interrupt processing

Interrupts are signals generated outside of the processor, halting the currently running application, to execute a set of instructions referred to as an interrupt handler or an interrupt service routine (ISR). This principle allows the development of event-driven applications that do not have to rely on polling. When a large workload is associated with an event, it is common to defer its processing to another task to avoid masking other interrupts. Since real-time systems must process incoming events in a timely fashion, the cost of interrupt handling must be minimal and very deterministic.

Interrupts are initiated by sending a signal on a specific line of an interrupt controller. The interrupt controller then redirects the signal to a pin of the processor to indicate that an interrupt request is pending. If the interrupts are not disabled on the processor, execution is halted and the appropriate ISR to execute located. Typically, the operating system or processor hardware finds the ISR by looking in an interrupt vector table that associates an ISR to a numeric identifier. The numeric identifier is provided by the interrupt controller that received the interrupt. Finally, the ISR is executed, the interrupt request cleared and the context of the next task to execute restored.

We define the interrupt latency as the delay between when an interrupt signal is sent and when is it acknowledged by the processor. The interrupt latency is influenced by two factors: the duration for which the interrupts are disabled by the operating system and the time required for the operating system to stop the current task, save its context and branch to the first instruction of the ISR. Measuring the length of the intervals where interrupts are disabled is not possible within a test scenario, since it can occur at any point within the operating system. However, the intervals can be measured if we trace the operating system, as will be discussed in section 4.5. The second aspect, which we call the interrupt processing time, can be measured in a test scenario.

TS.1 We propose to use software generated interrupts to measure the interrupt processing time without any external hardware. As shown in Figure 4.2, the test scenario is composed of a single task. The task triggers an interrupt in a loop.

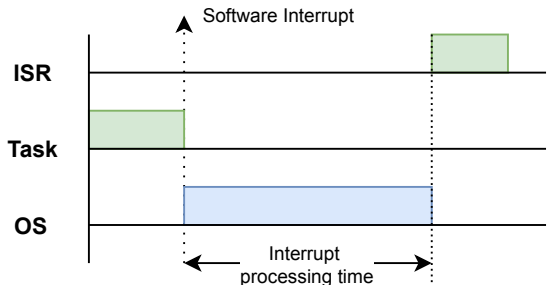


Figure 4.2 Test scenario to measure interrupt processing (TS.1)

4.4.2 Inter-task synchronization

Real-time systems are generally composed of multiple tasks that synchronize with each other through services exposed by the operating system. Since their use is common, the cost of these services must be well understood to build efficient applications. The reference implementation of the benchmark covers three services that were identified as the most common, since they are also targeted by the previous benchmarks presented in the introduction: cooperative scheduling, semaphores and mutexes. The structure of these tests could be reused with other synchronization primitives with minor modifications.

Cooperative scheduling

In cooperative scheduling, tasks willingly give up their place to other tasks. It is a very basic form of scheduling since tasks can only be put in two states by this service: running or ready for execution but preempted. When the operating system is called by a task that wants to relinquish execution to another one, it first saves the context of the task and then looks for the next task to be scheduled.

We identify a single metric for cooperative scheduling. The time it takes to switch the context from one task to another. This metric includes the time to save the context of the preempted task, the scheduling overhead to decide which task to execute next and the time to restore the context of the newly elected task. Saving and restoring the context should not be affected by the number of tasks on the system. However, depending on the implementation of the scheduler, the scheduling overhead could be higher when more tasks are active. To stimulate this variable, we propose a scenario that scales from 2 to N tasks.

TS.2 In this scenario, all the tasks share the same priority and start off from the same function. The core loop of each task is composed of a configurable workload followed by a call to yield the execution to another task. In the code leading up to the core

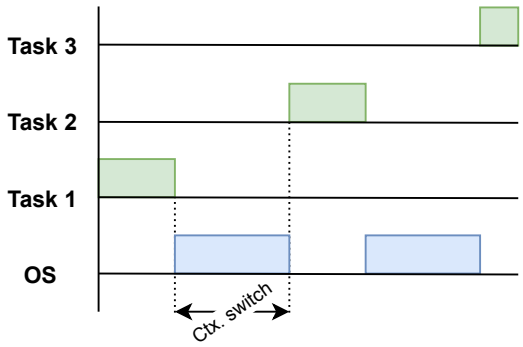


Figure 4.3 Test scenario to context switch time (TS.2)

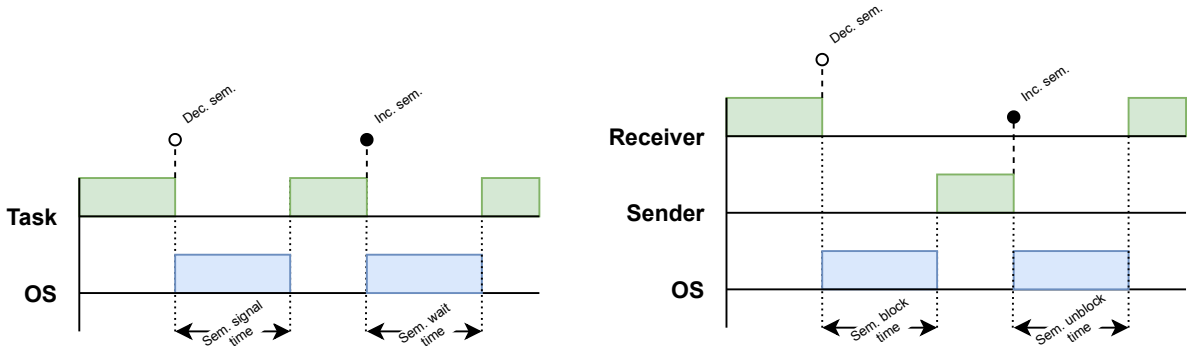
loop, each task reads a lock protected counter and increments its value. If the task is the first one to run, it executes a core loop where a timestamp is read before yielding. If the task is the second one to run, it executes a core loop where a timestamp is read after the return of the yield function. All other tasks simply call the yield function after executing their workload. Starting from the second iteration, the difference between these two timestamps will be the time it took to switch the context between two tasks. Figure 4.3 shows the execution of the scenario with three tasks.

If the number of active tasks handled by the operating system does have an impact, it would be observable in any of the context switches since all the tasks are always ready when a yield occurs. This scenario scales up correctly if the assumption that cooperative scheduling between tasks of the same priority is implemented in a first in first out manner. Otherwise, the values used to compute the time difference could come from different context switches. In that case, ordering can be enforced by creating only two tasks.

Semaphore

Semaphores are objects managed by the operating system and shared among tasks to synchronize their work. Tasks manipulate a semaphore by either incrementing its value or decrementing it. When a task decrements the value of a semaphore that has already reached zero, it is put in a waiting state by the operating system. The task will be woken up when another task increments the value of the same semaphore. Multiple semaphores can coexist on the system, each using its own wait queue. We establish four test scenarios to characterize the efficiency of the management of semaphores by the operating system.

TS.3 This scenario checks the times needed to increment the value of a semaphore when its wait queue is empty and to decrement its value when it is greater than one. The



(a) Semaphore processing time (TS.3)

(b) Semaphore block and signal time (TS.4)

Figure 4.4 Base test scenarios for semaphores

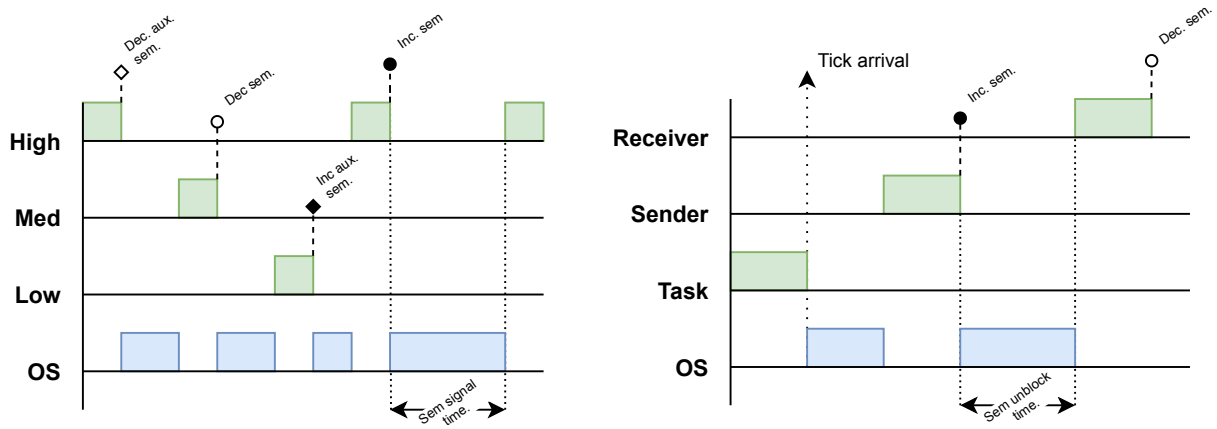
scenario is composed of a single task that increments and decrements the value of a semaphore in a loop, as shown in Figure 4.4 (a).

TS.4 This scenario records the time to switch the context to another task when a task blocks by decrementing a semaphore, or when it wakes up a higher priority task by incrementing a semaphore. Two tasks with different priorities are created in the test. The receiver task is instantiated with a higher priority and continuously decrements a semaphore. The sender task has a lower priority and always increments the value of the same semaphore. Figure 4.4 (b) displays the ordering of the tasks.

TS.5 This scenario evaluates the impact of waking up a lower priority task on the time to increment a semaphore. As Figure 4.5 (a) indicates, three tasks are created in the test. The highest priority task first waits on an auxiliary semaphore to let the medium priority task run. The medium priority task blocks while decrementing a second semaphore. Finally, the lowest priority task runs and wakes up the highest priority task by incrementing the auxiliary semaphore. The highest priority task then computes the time taken to increment the semaphore that the medium priority task is waiting on. At most one task can be woken up by incrementing the semaphore. Consequently, the test does not study the effect of a greater number of tasks in the wait queue of the semaphore.

TS.6 This scenario monitors the effect of a background workload on the time to switch from a sender task to a receiver task. The scenario is depicted in Figure 4.5 (b). Much like the scenario TS.4, there are two tasks of different priorities that are synchronized with the use of a semaphore. However, the sender task is periodically woken up by a timer instead of continuously running. Between two periods of the sender task, a

variable amount of tasks are working on a configurable workload. We only record the time required to wake up a new task, because this is the metric that is important to estimate the response time of an application. Tasks will block on a semaphore when no more work is available, so the time it takes is less of a concern.



(a) Semaphore signal time (TS.5)

(b) Semaphore signal time with workload (TS.6)

Figure 4.5 Advanced test scenarios for semaphores

Mutex

Mutex, short for mutual exclusion, are objects managed by the operating system and used by tasks to lock resources for exclusive use. When a task acquires a mutex, the operating system ensures that there is no other task locking it before letting the access go through. When the mutex is released, the operating system checks if another task was waiting to access it before letting the current task continue. The task releasing the mutex must be the same that acquired it earlier, otherwise locking semantics are not respected. A higher priority task can be blocked by a lower priority task if the latter one acquired the mutex before the higher priority task could. To reduce the latency of the higher priority task, operating systems commonly implement a priority inheritance protocol to temporarily elevate the priority of the task holding the mutex [90]. To test the performance of the handling of mutexes by the operating system, we propose three test scenarios.

TS.7 This scenario measures the time to lock and unlock a mutex that is not contented. The test scenario is composed of one task that acquires and releases a mutex in a loop. This scenario is illustrated in Figure 4.6.

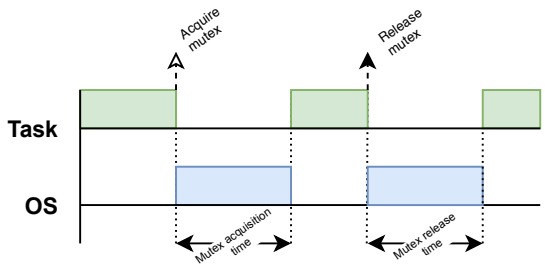


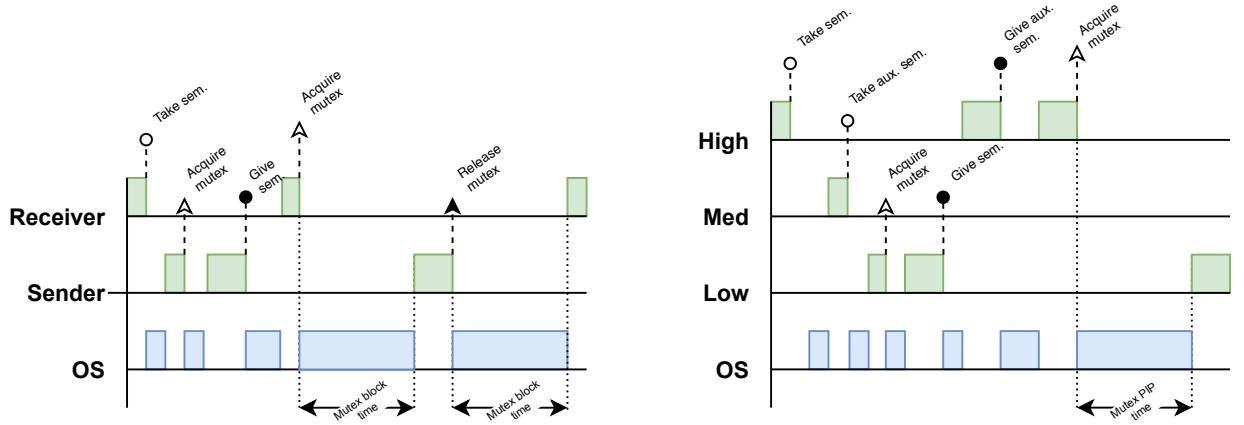
Figure 4.6 Mutex acquisition and release time (TS.7)

TS.8 This scenario records the times taken to block and to wake up a task when a mutex is contented. To artificially create such a situation, two tasks of different priorities are created. The receiver task has a higher priority and first blocks on a semaphore. The sender task, which has a lower priority, acquires a mutex and then increments the semaphore to wake up the receiver task. The receiver task then tries to acquire the mutex but is blocked since the lowest priority task has already acquired the mutex. The sender task will run again and release the mutex. Figure 4.7 (a) shows the execution of this scenario.

TS.9 This scenario is similar to the previous scenario but checks if priority inheritance incurs an additional cost on the mutex acquisition. Three tasks with different priorities are created. The highest priority task and the lowest priority task lock the same mutex. As displayed in Figure 4.7 (b), the medium priority task is set to the ready state by the highest priority task before it tries acquiring the mutex. If the operating system implements priority inheritance, the lowest priority task will be scheduled since it has a lock on a mutex that a higher priority task is waiting on. Otherwise, the medium priority task would run first.

4.4.3 Inter-task communication

Applications composed of multiple tasks will often require some means of communication to exchange data between tasks. To ensure that the data reaches its endpoint in the delay prescribed by the timing constraints, it is important to know the cost of such a mechanism. In this section, we describe test scenarios to test the performance of message queues.



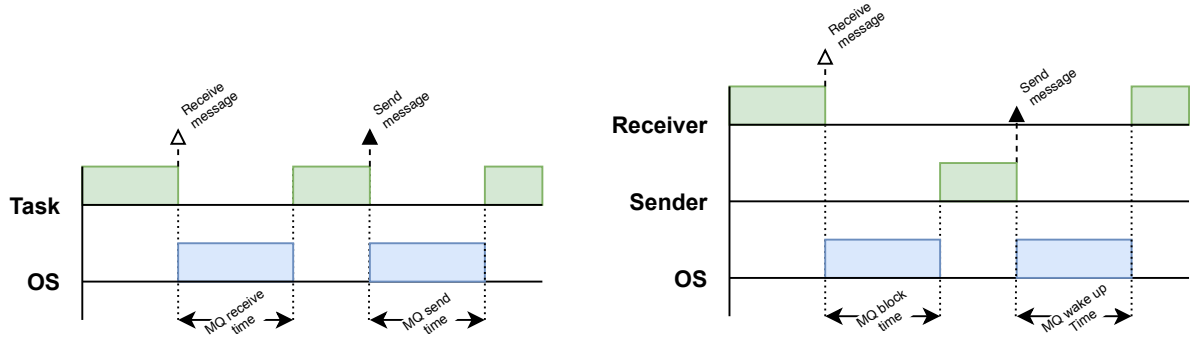
(a) Mutex block and wake up time (TS.8)

(b) Mutex acquisition with priority inheritance (TS.9)

Figure 4.7 Test scenarios for mutexes

Message queue

Message queues are a common implementation of an inter-task communication mechanism. They act as a buffer to retain a variable number of messages until they are consumed by a task. To obtain a fair comparison between the implementations of message queues, it is essential to configure them identically. The depth of the queue, the size of the messages and the way the data is transferred (by copy or reference) must be the same. The configuration is handled by the user of the benchmark. To assert the performance of an implementation, we propose the three following scenarios.



(a) Message queue send and receive time (TS.10)

(b) Message queue block and signal time (TS.11)

Figure 4.8 Base test scenarios for message queues

TS.10 This scenario measures the times needed to receive a message and to send a message. A task sends a message in a queue, then reads it. This test scenario is depicted in Figure 4.8 (a).

TS.11 This scenario records the times required to block a task that tries to receive a message on an empty queue and to wake up a higher priority task by sending a message. The scenario is composed of two tasks of different priorities. The lowest priority task is the sender task and the highest priority task is the receiver task. The sender task sends a message in the queue, which wakes up the receiver task. As indicated in Figure 4.8 (b), the length of the context switches are measured.

TS.12 This scenario looks at the effect of a background workload on the time to wake up a task, when a message is received in the message queue being waited on. The scenario is built in a similar manner as the second scenario, but the sender task waits on a timer before sending the message. When the sender and the receiver tasks are not active, a configurable background workload is scheduled.

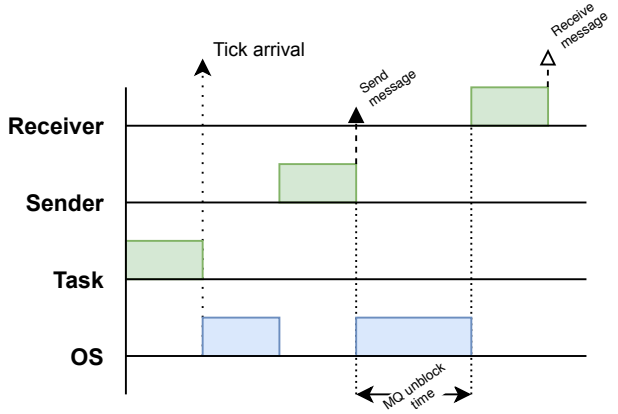


Figure 4.9 Test scenario for message queues with workload (TS.12)

4.4.4 Partition jitter

Operating systems with space and time partitioning have to efficiently implement the services discussed in the previous sections and are also required to provide strong isolation guarantees. A major concern will be staying within the time slots assigned to the partitions. The operating system has to ensure that the actual time used by the partitions is no more and no less than what was requested. If an application has less time to execute than expected, it could fail to fulfill its workload before the end of its deadline. If a partition executes for

longer than requested, the response time of applications residing in other partitions will be longer than anticipated since their scheduling period will be bigger.

Measuring the length of time for which each partition is allowed to run is difficult from user applications since they have no knowledge of when their partition is scheduled out. We can, however, observe if drifting occurs in the expected period, from the partition itself, with the following test scenario.

TS.13 In each partition, a task with the highest possible priority is created. This task waits on a timer with a period that is greater than the execution time of the partition but smaller than the period of the partition itself. Thus, this task should be ready whenever the containing partition starts executing. Each time this task runs, it records the time at which it was woken up and the time at which it expects to be woken up next. The next time this task runs, it computes the difference between these two values. The result acts as an indicator of the jitter in scheduling of the partitions. The greater the difference between the expected time and the actual time, the worse the operating system is at enforcing the schedule. This test scenario can be combined with any scenarios previously described to study the effect of a workload on the partition jitter. Figure 4.10 displays the execution of the test scenario.

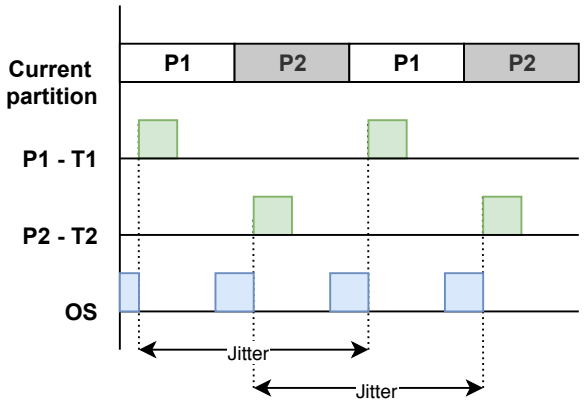


Figure 4.10 Test scenario for partition jitter (TS.13)

4.5 Integration with tracing

If surprising results are observed while running the benchmark, identifying their cause is essential. These problems could arise from a misconfiguration, in which case they could be fixed, or from limitations of the operating system. Tracing the operating system gives additional context to the results, allowing the user to explore the states of all the tasks,

while the benchmark was executing. However, finding the exact moment when a problem occurred can be difficult, manually looking through a trace that spans the whole execution of a test scenario which collected measures over many iterations. Therefore, we propose in this section an optional tool that integrates with the benchmark to analyze the results and identify the points in time where outlier values were measured. This tool is a plugin to the open source software Trace Compass [2]. It is composed of two sets of views, which are freely available [3]. The first group of views is created especially for operating systems with space and time partitioning. The second group of views helps the analysis of the results of the benchmark.

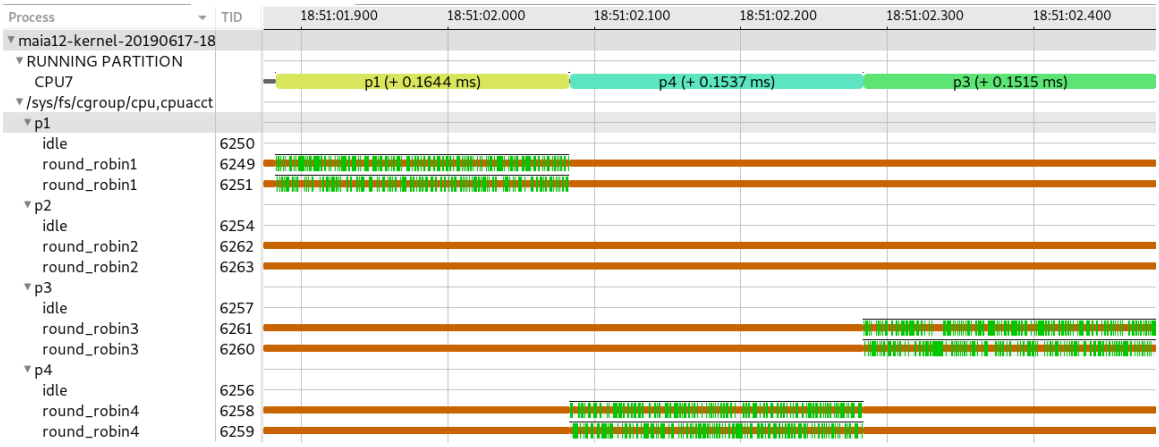


Figure 4.11 View in Trace Compass to analyze traces from space and time partitionning operating systems

The objective for the first view is to adapt trace visualization to the needs of operating systems with space and time partitioning. We developed a view that clearly shows to the user the schedule of the partitions on each core and how closely they follow the expected schedule. The view is a timegraph that hierarchically lists the tasks and the related partitions on the Y axis. On the X axis, the states of the tasks are displayed. The tasks are either running, preempted, interrupted or waiting for an event to occur. A color is associated to each state. This view is pictured in Figure 4.11. The analysis can also automatically compute the time difference between the actual execution time of the partitions and their expected execution time, if it has access to the schedule. When this option is activated, the labels of the partitions in the view include this time difference. For this analysis to work correctly, the user has to identify in the trace the beginning of the scheduling period. To populate the view, the analysis requires information about the scheduling of the processes within the

2. `eclipse.org/tracecompass`
3. `github.com/gchamp20/ARINC653-TC`

partition and information about the scheduling of the partitions. We discuss in more detail how this information can be collected while using the ARINC 653 scheduler of Xen or when mimicking a space and time partitioning operating system on Linux, in section 4.7.

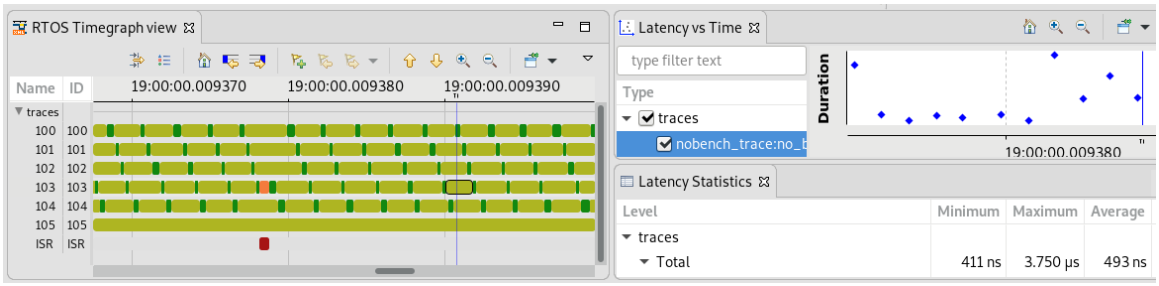


Figure 4.12 Views in Trace Compass to analyze the results of the benchmark

The goal of the views supporting the results of the benchmark is to point out occurrences of outlier values. In a normal execution of the benchmark, timestamps are read at precise times before an action of interest takes place. When the benchmark is executed for use with tracing, instead of reading a timestamp, the application produces an event logged by a tracer. Each test scenario uses specific event types so that the view recognizes automatically which metrics are available in the trace. When a trace containing the correct information is loaded into Trace Compass with the benchmark plugin activated, a view summarizing the results of the test in a table, or displaying them as a scatter plot, is shown. These views are shown in Figure 4.12. The view displaying the maximum, minimum and average length of the intervals can be used to pan the timegraph automatically to the problematic occurrences. The results are calculated by using the XML scripting capabilities of Trace Compass [65]. A custom state machine described in XML consumes the events in the trace and populates an internal data structure of Trace Compass called the *State System History Tree*. We connect default views of Trace Compass to our custom state system to create those views, enabling us to quickly analyze the results of the benchmark.

For minimalist real-time operating systems that do not have a mature tracing infrastructure, the benchmark also includes a very basic tracer implementation. This tracer can be connected to hooks that are often already present in the operating system. The tracer will record the information in the format expected by the trace analysis tool. To make the analysis framework of Trace Compass available to operating systems that do not support space and time partitioning, a simpler timegraph to show the state of the tasks was developed. This timegraph is shown on the left of Figure 4.12.

4.6 Intra-partition benchmarking

In this section, we present the results gathered by executing the benchmark on three real-time operating systems that do not implement space and time partitioning. Thus, the results only relate to the test scenarios presented in sections 4.4.1 through 4.4.3. The hardware used as well as the configuration of the selected operating system is presented in section 4.6.1. The results are discussed in section 4.6.2

4.6.1 Test environment

The selected platform for the tests is the Raspberry PI 2B+. The Raspberry Pi 2B+ uses a Cortex A7 quad-core processor clocked at 900MHz. Each core on the chip has a 32KB L1 cache and shares a 512KB L2 cache with the other cores. The cores have access to 1GB of RAM memory.

We tested three open source operating systems with the benchmark we developed: FreeRTOS v10.1.1, RTEMS v4.11.3 and Linux v4.14 with the Raspbian Stretch Lite distribution. Indeed, we found no ARINC 653 compliant RTOS available for publishable benchmarking, due to licensing restrictions. The tests were run with the *force_turbo* parameter set to true in the Raspberry Pi boot configuration file, to ensure that the cores always run at their maximum frequency. For FreeRTOS and RTEMS, the data cache, the instruction cache and the MMU were enabled in the startup code. Since these two operating systems do not interact with the memory management unit (all tasks share the same address space), we set up the MMU with a direct mapping between the virtual addresses and the physical addresses. For Linux, cores 3 and 4 were isolated to execute the tests using the *isolcpus* Linux boot argument. Threads will only get scheduled on an isolated core when their affinity is explicitly set to its value. The threads created by the test scenarios use the *SCHED_FIFO* scheduling policy to ensure that we can control their priority. To collect the traces on FreeRTOS, we use the basic tracer provided with the benchmark. On RTEMS, we use the trace capture engine built into the operating system. For Linux, we use the LTTng kernel tracer v2.10 [49].

4.6.2 Results

This section presents the results of the benchmark when executed on FreeRTOS, RTEMS and Linux. All the results are expressed in nanoseconds. On Linux, the time measurements were taken using the *clock_gettime* system call. On RTEMS and FreeRTOS, the timestamps were taken by reading the value of the cycle counter register (*PMCCNTR*) directly. The clock cycles reported by the benchmark were converted to nanoseconds for the results presented

here.

Interrupt processing

FreeRTOS and RTEMS let user applications provide custom interrupt handlers. These two operating systems have in their public interface a function to map a new interrupt handler to an interrupt number. On Linux, this is restricted to kernel modules and drivers. Therefore, interrupt processing time was not measured on Linux since the structure of the benchmark assumes that any application can install a new interrupt handler. To generate a software interrupt on the Raspberry PI, we choose to use one of the four per core mailbox available [91]. A core mailbox interrupt can be triggered by any application that writes to the core mailbox write-set register.

Table 4.1 presents the time it takes in FreeRTOS and RTEMS to process an interrupt. The interrupt processing is the interval from when the interrupt is triggered by writing to the core mailbox register to when the user provided interrupt service routine is entered. RTEMS takes about twice as much time on average to reach the ISR. A possible explanation for this variation is the difference in the scope of both operating systems. The interrupt handling code of RTEMS includes handling for multicore systems, although the Raspberry Pi port does not make use of multicore. The code uses memory barriers and keeps a per CPU interrupt nesting counter. FreeRTOS has no port usable on multicore processors, thus its interrupt handling code is simpler. It does not use memory barriers and only keeps track of one interrupt nesting counter.

Cooperative scheduling

This section presents the results of the cooperative scheduling scenarios. FreeRTOS supports cooperative scheduling among tasks of the same priority by calling a special function. Linux provides a similar interface. On the other hand, RTEMS has no dedicated function for yielding. Instead, the function to delay the execution of a task has to be called with a special

Table 4.1 Results of the interrupt processing scenario (in ns, $N = 5000$)

Metric	FreeRTOS			RTEMS		
	Mean	SD	Max	Mean	SD	Max
Processing	636	68.37	952	1217	86.66	1537

Table 4.2 Context switch time in the cooperative scheduling scenarios (in ns, $N = 5000$)

Tasks	FreeRTOS			RTEMS			Linux		
	Mean	SD	Max	Mean	SD	Max	Mean	SD	Max
2	170	21.82	1143	484	25.71	1625	2788	893.41	58229
10	170	21.96	1104	531	34.49	1715	2982	843.28	53229
30	171	32.70	1205	588	59.16	1876	3179	1269.40	66979
50	172	45.35	1324	614	56.14	1858	3574	720.17	29219

flag to indicate that a yield is requested.

Table 4.2 shows the mean, standard deviation and the maximum value observed for the context switch time, depending on the number of tasks created by the scenario. We notice that FreeRTOS takes the lowest amount of time on average to perform this context switch. Linux takes the most time, which is understandable since it is the only studied OS that performs a switch between user mode and kernel mode. Therefore, Linux has to manage the memory address spaces when entering the kernel, whereas FreeRTOS and RTEMS do not.

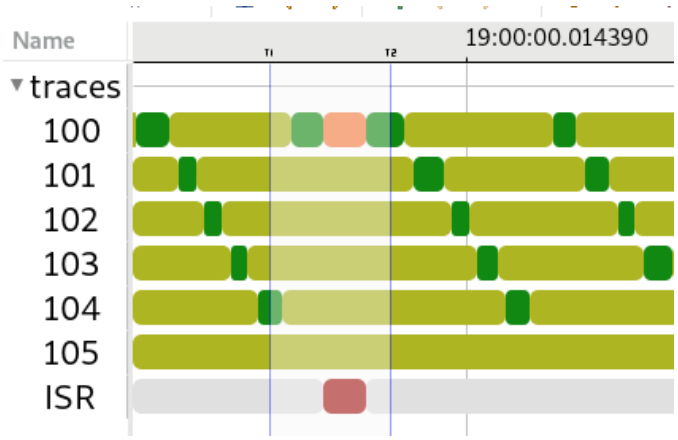


Figure 4.13 Worst case execution time for FreeRTOS visualized in Trace Compass

In all three operating systems, we see that the maximum value observed is much higher than the average value. To explain these results, we traced the execution of the benchmark and used the visualization tool suggested in section 4.5. Figure 4.13 shows the problematic execution for FreeRTOS located by the trace visualization tool. When an interrupt occurs before the operating system starts switching the context, the action is delayed until the handling of the incoming interrupt is completed. When this situation occurs, the delay

measured is much higher than the time it actually takes for the operating system to complete the context switch. This has a lower probability of occurring in a normal application but is an interesting result since it can affect the worst-case execution time. We arrive at the same conclusion for RTEMS by looking at its execution trace. Looking at the trace from the Linux execution, we see the same scenario occurring, but the base time when it is not interrupted has more variability than the two other operating systems.

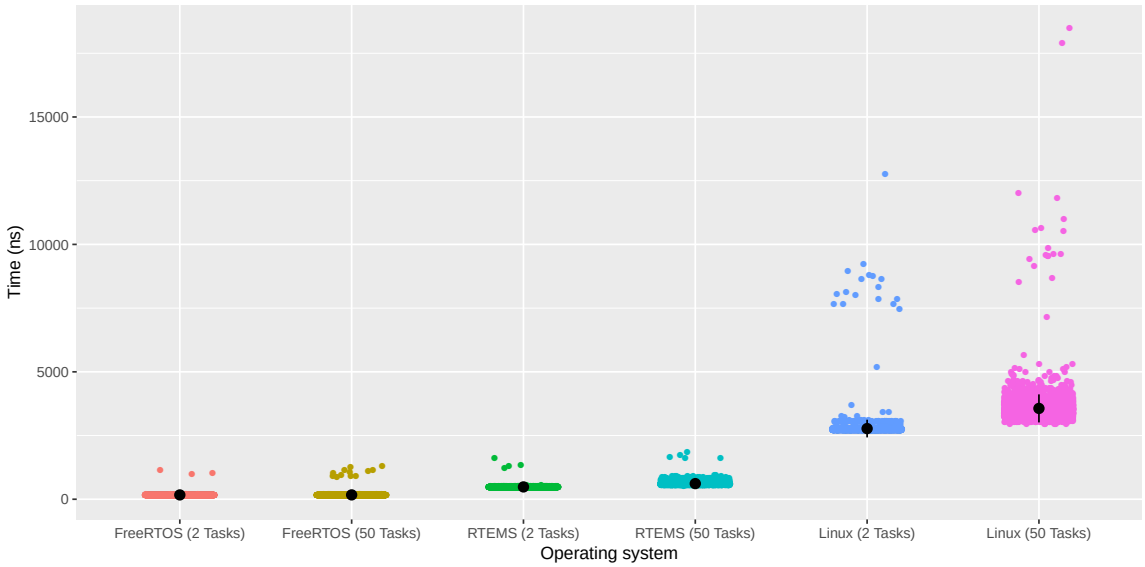


Figure 4.14 Distribution of the context switch time samples

Another interesting conclusion from these results is that the number of tasks definitely has an impact on the overhead of the scheduler for RTEMS and Linux. Figure 4.14 presents the distribution of the samples for the test scenarios with 2 tasks and 50 tasks. For Linux, three extreme values, higher than 20000 nanoseconds, are not presented in the graph. The black dot on each distribution is the mean and the vertical bars are the length of the standard deviation. While FreeRTOS maintains the same average, it is clear that the mean time and the spread of the distribution are greater when the number of tasks is higher on RTEMS and Linux.

Semaphore

This section discusses the results gathered by executing the semaphores scenarios. On FreeRTOS and RTEMS, semaphores can be created as binary or counting semaphores. The value of binary semaphores is limited to one, but counting semaphores can be incremented up to a configurable limit. On Linux, the semaphores created through the POSIX API can only be

counting semaphores. Consequently, the benchmark was configured to also create counting semaphores for FreeRTOS and RTEMS

To get a more accurate picture of the determinism of each operating system when handling semaphores, the top ten highest outlier data points were rejected before computing the results shown in table 4.3. As mentioned in the previous section, when a tick interrupt occurs, the result is much higher than the actual time taken by the operating system to perform the action.

Incrementing or decrementing a semaphore when its wait queue is empty takes a comparable amount of time for the three operating systems. Linux reaches a performance similar to the other operating systems since, in that situation, it does not have to switch to kernel mode. However, the scenario that checks the effect of waking up a lower priority task by incrementing a semaphore makes the average time of Linux jump up to 3305 nanoseconds. By tracing the execution, we notice that incrementing the semaphore now requires a *futex* system call, which makes the execution about 10 times longer.

We also notice that the standard deviation of the results of the test scenarios to increment a semaphore or block on a semaphore is quite larger than in the other scenarios. Looking at the trace generated during a second run, we make similar observations but do not see any interrupts that could affect the results. However, when a context switch is performed because the semaphore was incremented, the results are much more deterministic.

Mutex

This section presents the results obtained while executing the mutex test scenarios. Mutexes are available in all three studied operating systems. By default, FreeRTOS and Linux mutexes will be held with priority inheritance activated. However, on RTEMS, priority inheritance is opt-in. For a fair comparison, in all test scenarios the mutexes used on RTEMS are created with priority inheritance activated. Table 4.4 presents the results. For these results, the top ten outlier data points were also rejected.

The acquisition and the release of mutexes has a similar cost to incrementing or decrementing a semaphore for the three operating systems. The times to block on a mutex or to wake up a new task by releasing it are also comparable to the time to block on a semaphore, or to wake up a new task by incrementing a semaphore, on FreeRTOS and RTEMS. This result is not surprising since mutexes can be thought of as binary semaphores. In FreeRTOS, the mutexes are acquired through the same interfaces as semaphores are incremented. In RTEMS, the mutex acquisition and semaphore incrementation code paths are different, but they both end

Table 4.3 Results of the semaphore benchmarking scenarios (in ns, $N = 5000$)

Metric	FreeRTOS			RTEMS			Linux		
	Mean	SD	Max	Mean	SD	Max	Mean	SD	Max
Inc.	249	76.09	472	119	6.00	148	308	19.51	416
Decrem.	103	0.18	104	100	2.52	124	310	18.46	365
Decrem.[1]	245	0.94	278	317	3.71	329	3279	76.84	4323
Wake up	475	2.02	482	663	4.70	692	7041	1233.21	10053
Wake up [2]	447	2.04	451	845	10.50	889	11884	2175.35	29218
Block	1074	77.73	1312	729	5.38	740	5893	958.11	13073

[1] Decrementing with lower priority task moved to ready.
[2] Wake up with 50 tasks background workload.

Table 4.4 Results of the mutex benchmarking scenarios (in ns, $N = 5000$)

Metric	FreeRTOS			RTEMS			Linux		
	Mean	SD	Max	Mean	SD	Max	Mean	SD	Max
Acquire	257	76.67	485	143	1.17	158	420	34.53	1042
Release	161	0.52	162	201	1.35	216	365	19.42	625
Wake up	568	6.51	579	764	4.80	959	9281	960.99	26666
Block	1111	8.83	1144	778	1.90	784	7418	868.51	24948
PIP	1113	9.34	1164	1059	4.65	1092	10429	1476.15	30521

up managing the wait queue with the same implementations. Therefore, it is reasonable to observe that both services have the same average cost.

On FreeRTOS, the scenario where the priority inheritance protocol has to be used does not increase the duration of the context switch that occurs when blocking on a mutex. On RTEMS and Linux, we notice that the average time required goes up by around 40%. However, the average time in the scenario with priority inheritance ends up around the same value for FreeRTOS and RTEMS. This is probably an implementation choice on the side of FreeRTOS to make the cost of the service more predictable in all possible scenarios.

Message queue

In this section, we analyze the results of the message queue test scenarios. On FreeRTOS and RTEMS, message queues are very similar. They are used to exchange messages of configurable size between tasks living in the same address space. On Linux, message queues are a mechanism used for interprocess communication. Consequently, on Linux, message queues have to handle transmitting data to different address spaces. To do so, message queues are created and opened through a filesystem. Considering this, we fully expect Linux to be slower to send messages than the two others. Table 4.5 summarizes the results. Like in the previous sections, the top ten outlier data points were rejected.

As we expected, the time to send a message or receive a message on Linux is the highest of all three operating systems tested. Since it has to handle inter-process communication, it cannot be optimized like semaphore or mutexes to be handled in userspace when there is no other task in the wait queue.

Like in other scenarios, we observe that FreeRTOS is the more deterministic for the actions

Table 4.5 Results of the message queue benchmarking scenarios (in ns, $N = 5000$)

Metric	FreeRTOS			RTEMS			Linux		
	Mean	SD	Max	Mean	SD	Max	Mean	SD	Max
Send	305	77.73	528	254	33.86	1340	2835	577.70	19010
Receive	181	3.22	191	224	29.15	1263	2699	421.76	17239
Signal	576	4.85	593	784	9.42	818	11876	1030.56	29739
Signal[1]	558	9.71	579	905	15.98	1012	13446	3366.47	59583
Block	1058	77.72	1310	844	9.69	879	8303	776.35	23333

[1] Signal with 50 tasks workload

contained in the scenarios. The effect of the background workload is minimal on the average time, whereas we observe a slight increase in the average cost to wake up a task with a message on RTEMS and Linux, with a 50 tasks background workload.

4.7 Inter-partition benchmarking

In the section, we present the results obtained by the benchmark for Linux and the Xen Hypervisor when configured to approximate real-time operating systems with space and time partitioning. The analysis studies the effect of a workload on partition jitter and the effect of multicore interference on the results of the benchmark. Section 4.7.1 presents the operating systems configuration and the hardware used. Section 4.7.2 discusses the results.

4.7.1 Test environment

To compare the performance of Linux v4.13.3 and Xen 4.13 in the context of space and time partitioning, we use different hardware than in the previous section. The tests are ran on a desktop computer using an Intel Core i7-4740 clocked at 3.6Ghz with 32G of RAM. The configurations and the software used to gather operating system traces for the systems are discussed in the subsections below.

Linux

Linux is highly configurable and can be tuned for real-time contexts. With the right configuration, it comes close to a space and time partitioning operating system, albeit not fully compliant with the ARINC 653 standard. To isolate tasks and deterministically limit their execution time, we propose to use CPU isolation and control groups.

CPU isolation can limit which tasks are eligible to run a set of CPUs. CPU isolation can be configured at boot time using the *isolcpus* boot parameter to indicate which CPUs cannot run tasks that have not explicitly set their affinity to these cores. For the analysis presented in this section, we isolate CPUs 2,3,6 and 7, which correspond to the physical cores 2 and 3.

Control groups are logical containers of tasks which are related to at least one controller. The controllers expose to the user a number of configurable parameters through a file system. To temporally isolate tasks, we propose to create one control group per partition, each attached to the CPU controller. When the kernel is compiled with the *CONFIG_RT_GROUP_SCHED* parameter, this controller exposes the *rt_runtime_us* and the *rt_period_us* parameters. By setting these parameters, we can control how long each par-

tition runs per time period. To ensure the ordering in the schedule with multiple partitions sharing the same CPUs, we insure that all the tasks within the same partition have priorities that are higher than the tasks in the partition they must precede. In each partition, an idle task with the lowest priority available picks up any time left unused by other tasks.

To collect kernel traces, we use the LTTng kernel tracer [49]. By default, the LTTng tracer does not produce information about which thread belongs to which control group. To work around this, we run a userspace application that scans the content of the control group file system to report which threads belong to which partition. This user space application only has to run once when all the threads for the test scenario have been created and is only used to investigate the results with trace analysis. The data presented in the tables is gathered from the benchmark output, without the LTTng tracer or the userspace application running.

Xen

Xen is a bare metal hypervisor used to concurrently run different operating systems. It uses the concept of domain to encapsulate the different operating systems it has to schedule. In every system using Xen, there is a *dom0* domain which has the highest privilege level of all. This domain is used to configure the system and create new domains. Xen offers a few scheduling policies to schedule the domains, including an ARINC 653 policy [17]. To use this scheduling policy for a set of domains, they must all belong to the same CPU pool containing a single CPU.

For the study, we use a *dom0* domain running the Ubuntu distribution with Linux v4.18. The domains scheduled with the ARINC 653 scheduling policy are running the Debian Stretch distribution with Linux 4.9.

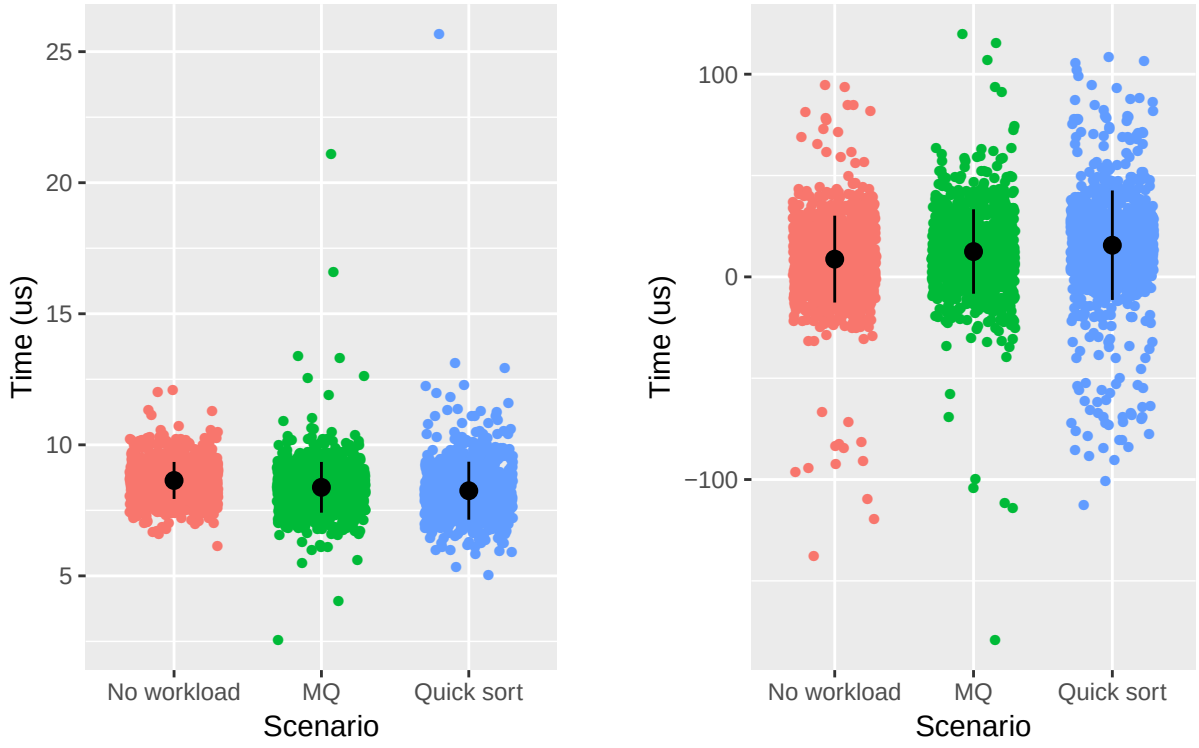
The kernel traces for each domain are created by the LTTng kernel tracer. This gives us information about the processes within each domain. To visualize the ordering of the partitions on the CPU, we use the traces from the Xentrace tool running from the *dom0* on the CPUs not used by the ARINC 653 domains. Since the domains and the hypervisor timestamps are not synchronized, we start a userspace application in each ARINC 653 domain that periodically performs an hypercall with a unique ID. This unique ID appears in both the LTTng traces and the hypervisor trace, allowing us to synchronize the traces using the Trace Compass trace synchronization feature. This application, Xentrace and the LTTng tracer are only executed to get insight into the results, not to generate the data presented in the tables, which is generated directly from the benchmark output.

4.7.2 Results

Partition jitter

Partition jitter is estimated using the test scenario proposed in section 4.4.4. The test measures the difference between the expected time and the actual time at which the partitions were scheduled.

To evaluate the capacity of the operating system to temporally isolate the applications, we run the benchmark under three sets of conditions. First, we gather the results of the benchmark when the system is not under any additional load. Then, we observe the effect of a background workload within the partition on its jitter. The background workloads are test scenarios from the benchmark. The chosen background workload are the message queue test scenario, that evaluates the block time and the wake up time (TS.11), and the cooperative scheduling test (TS.2) scenarios with 5 tasks running the large quick sort application from MiBench [92]. For the test, the partition containing the workload has 100 milliseconds of execution time per period of 500 milliseconds.



(a) Partition jitter on Linux

(b) Partition jitter on Xen

Figure 4.15 Partition jitter of the tested systems ($N = 1000$)

Table 4.6 Results of the partition jitter scenario (in us, $N = 1000$)

Metric	Linux			Xen		
	Mean	SD	Max	Mean	SD	Max
No workload	8.641	0.71	12.096	8.760	21.43	94.629
MQ	8.379	0.96	21.107	12.510	20.83	119.649
Quick sort	8.250	1.11	25.659	15.626	27.00	108.688

Figure 4.15 presents the measured jitter during 1000 iterations of the test scenario under the three sets of conditions for both systems. The average jitter of each distribution is displayed as a black dot and the standard deviation around the mean is represented by black vertical lines. Table 4.6 lists the mean, the standard deviation and the maximum values of the distributions.

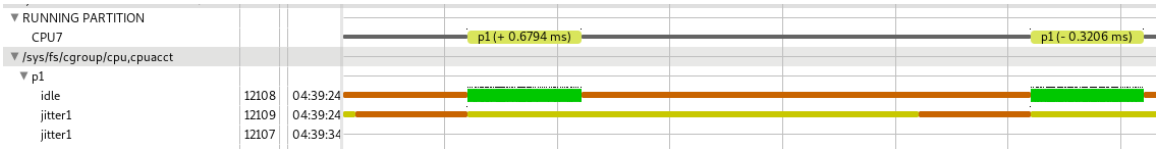


Figure 4.16 Linux partition jitter execution viewed in Trace Compass

For Linux, the workload running inside the partitions has no discernible effect on the jitter of the partitions. Since partitions are emulated through control groups, the execution is very similar to when any other process is woken up, as shown in Figure 4.16. Control groups do not enforce spatial partitioning, therefore the operations required to switch between them are no different from a context switch between threads of different processes. The execution trace shows a particularity of the approach to imitate space and time partitioning with control groups. Even when the partition has consumed all of its execution time, tasks contained within the control group can still change state.

On Xen, the results of the partition jitter test are much more dispersed. We notice that in quite a few occurrences, the monitor task runs before it expects to, giving a negative result. Since the monitor task is always the first one to run in the partition, this is an indicator that the ARINC 653 scheduler of Xen has a tendency to let the partitions run for less time than specified. The results also show that the workload has an effect on the average jitter and the standard deviation. The quicksort workload, which loads around 600KB of data to sort the values has the biggest impact.

Multicore interference

This section explores the effect of interference between unrelated concurrent partitions when using a multicore processor on the results of the benchmark. To assert if concurrent partitions can affect the performance of operating system services, four partitions are created. The first two partitions run on the same CPU. One of these partitions runs the cooperative scheduling scenario (TS.2) of the benchmark and the second one runs an idle process. The second two partitions run on a different CPU. One of the partitions also runs the cooperative scheduling scenario configured to run the large quicksort application of MiBench. Consequently, each time one of the tasks runs, it performs a quicksort on 150000 integers, taking at least 600 kilobytes of memory. This data does not entirely fit in the 256 kilobytes of L1 cache available per core. Figure 4.18 presents the baseline results when no other partition is running, when partitions are running on different physical cores, and when partitions are running on the same physical core with hyper-threading.

The time it takes to switch the context of a task in the cooperative scheduling scenario is not significantly affected by a partition running on another physical core. Since the metric measured does not require new data to be loaded or cached, or cache management operations since tasks within partitions share the same address space, a concurrent application causing cache misses in the L2 cache does not have a great influence. However, the execution speed of both applications could be affected if they were competing for space in the L2 cache. To illustrate this, we can look at the results when both partitions are running on the same core. We see in Table 4.7 that the standard deviation noticeably increases when using hyper-threading to run the concurrent partitions.

Figure 4.18 shows the view developed for Trace Compass with a trace for Xen. We immediately see that as soon as the concurrent partition starts running on the same core, the interval measured for the context switch time goes up. This explains the two sets of points in

Table 4.7 Cooperative scheduling scenario with a multicore workload (in ns, $N = 10000$)

Metric	Linux			Xen		
	Mean	SD	Max	Mean	SD	Max
Baseline	661	40.39	3973	2924	102.68	6240
Physical	662	37.11	2474	2940	310.31	13974
Hyp.	679	51.86	3126	3439	530.20	19811

Figure 4.18 for the hyper-threading scenario on Xen. This investigation shows the utility of trace visualization to quickly identify sources of slowdowns. The proposed benchmark is an example of an application that can be instrumented to populate this view, but the interval measured could come from any application.

In summary, we completed two separate experiments to test the viability of the benchmark. In the first experiment, we compared the performance of FreeRTOS, RTEMS and Linux without space and time partitioning. This shows that the benchmark can be used to characterize the performance of operating system services within a partition. In the second experiment, we compared the performance of Linux configured to approximate a system with space and time partitioning and Xen with the ARINC 653 scheduler. By analyzing the results of the benchmark, we conclude that Linux control groups are better at enforcing their scheduling period, but we should keep in mind that the configuration used is very specific and less obvious to set up than Xen.

4.8 Limitations

The proposed benchmark is a tool to quickly collect data about the performance of an operating system. It aims to covers metrics that fall in the critical path of real-time applications, but since the metrics are measured in controlled scenarios, they should never be considered as the maximum upper bound. They should be used as a baseline for comparing two operating systems or to have an estimate of the expected performances. Also, the proposed scenarios only cover services that are available in most operating systems. With very little additional work, the structure of the scenarios could, however, be reused to test other services.

The view for space and time partitioning operating systems is an important contribution towards making trace analysis for such systems more widely available but could be further enhanced. The view displays the ordering of partitions on the CPUs but lacks information about the scheduling of tasks on the virtual CPUs used by the partitions themselves. Also, the view has no clear indicator of the priority of the tasks running.

4.9 Conclusion and Future Work

In this paper, we explored the use of operating system tracing to identify the sources of performance issues in real-time operating systems with space and time partitioning. We developed a trace analysis plugin for the open source software Trace Compass to visualize the scheduling of partitions on CPUs as well as the state of the tasks throughout time. To

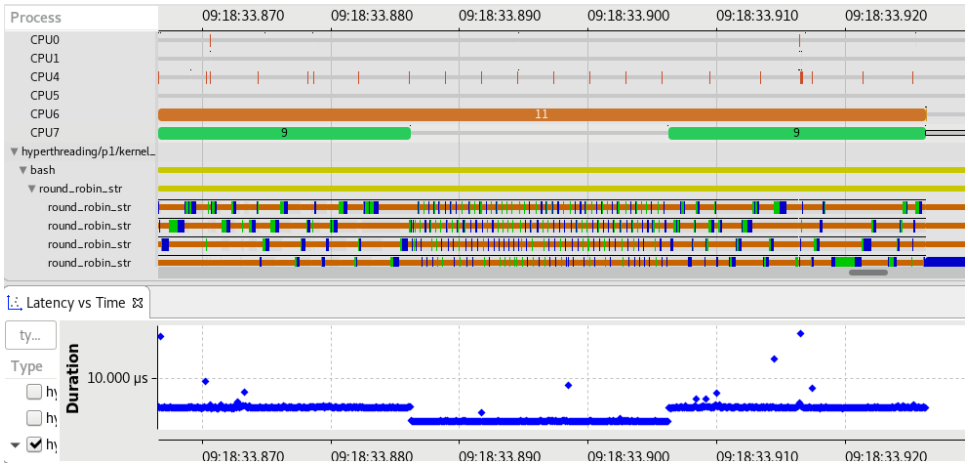
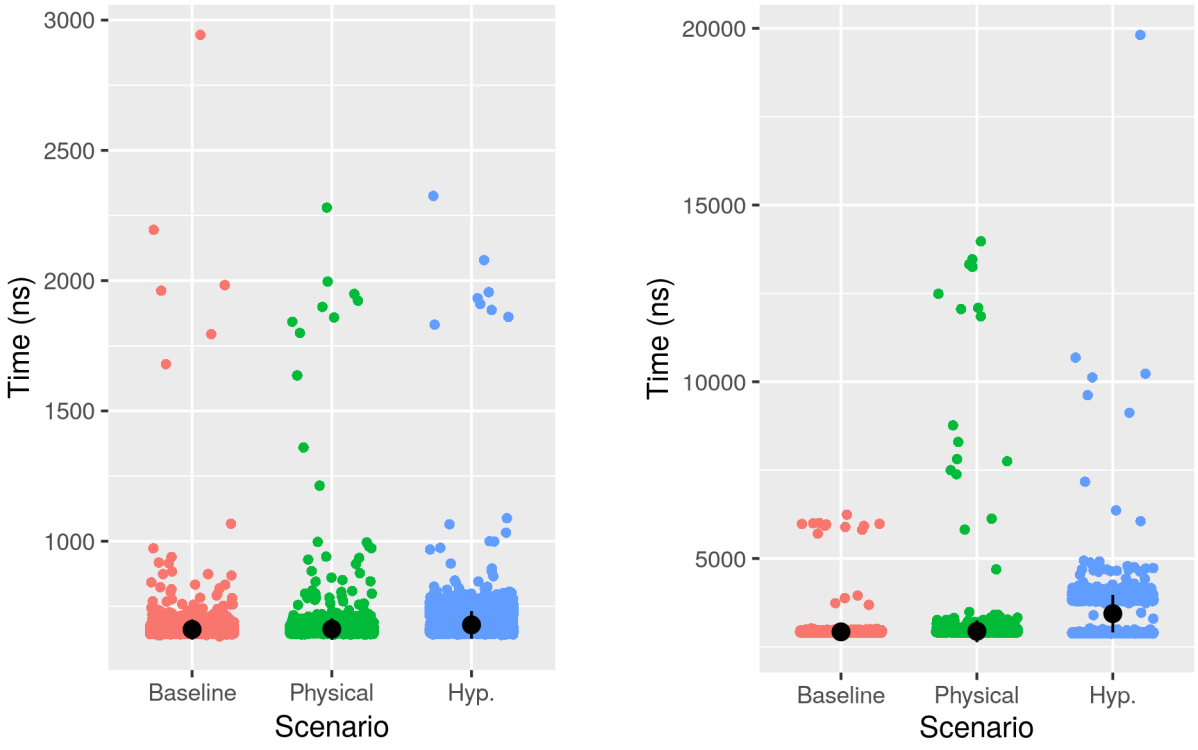


Figure 4.17 Xen cooperative scheduling results viewed in Trace Compass



(a) Results for Linux

(b) Results for Xen

Figure 4.18 Cooperative scheduling results with multicore workload ($N = 10000$)

validate the utility of the visualization tool, we proposed RTOSBench [4], a new open source and portable benchmark that, to our knowledge, is the first especially created for operating systems with space and time partitioning. This benchmark, as well as the trace analysis tool, were used to analyze the performance of two sets of operating systems.

Future work on the topic of trace analysis could investigate the integration of values from performance counters on the trace visualization timegraph. As we showed in the last section of the paper, concurrent applications can influence each other if they share the cache. Displaying the value of the cache misses performance counter directly in the view would help developers find potential interference problems more quickly. Another avenue of interest is to study the performance of inter-partition communication in the applications. The ARINC 653 standard defines an interface to send messages to other partitions. The trace analysis plugin developed could be augmented to provide detailed information on the latency in the transmission of messages.

Conflict of interest

The authors declare that they have no conflict of interest.

4. `github.com/gchamp20/RTOSBench`

CHAPITRE 5 DISCUSSION GÉNÉRALE

Dans ce chapitre, nous faisons un retour sur les résultats de l'étude présentée dans le chapitre 4 et approfondissons certains d'entre eux.

5.1 Retour sur les résultats

5.1.1 Visualisation de traces de systèmes temps réel avec partitionnement spatial et temporel

Les outils présentés dans la revue de littérature ne conviennent pas pour l'analyse de traces provenant de systèmes d'exploitation temps réel avec partitionnement spatial et temporel. Aucun d'entre eux ne propose une vue permettant de visualiser l'ordonnancement des partitions sur plusieurs coeurs. Dans l'article du chapitre 4, nous introduisons un plugiciel au logiciel Trace Compass qui comble ce manque. L'outil développé fonctionne actuellement avec des traces provenant de Linux ou de Xen. Dans le cas de Linux, la trace doit contenir une image du contenu du système de fichier de groupe de contrôle afin que l'analyse ait l'information nécessaire pour relier les tâches à leur partition. Pour l'étude, ces informations sont créées par une application qui parcourt ce système de fichier en créant des évènements enregistrés par le traceur LTTng dans l'espace usager. Sur Xen, une trace de l'outil Xentrace doit être fournie. Puisque les traces provenant des domaines et celle de Xentrace ne sont pas synchronisées, une application de synchronisation doit être exécutée dans les domaines. Dans notre cas, l'application effectue un *hypercall* demandant le numéro de version de Xen. Un argument supplémentaire contenant un numéro d'identification est ajouté à l'appel. Cet appel est choisi puisque contrairement à la majorité des *hypercall* possibles, il n'a aucun effet secondaire sur la configuration du système. Pour que l'identificateur unique soit présent dans la trace de LTTng, l'application exécutée dans le domaine crée un évènement avant d'effectuer l'*hypercall*. Par défaut, le point de trace pour l'*hypercall* du numéro de version n'enregistre pas l'argument supplémentaire. Xen est donc recompilé avec le point de trace modifié pour inclure ce paramètre. Par conséquent, l'utilisation de l'outil demande un minimum de configuration au préalable. Toutefois, du travail pourrait être fait afin d'améliorer l'environnement de collection de traces pour le rendre plus simple à utiliser.

Dans l'article, nous avons utilisé l'outil de visualisation afin d'analyser les résultats du banc d'essai. Ceci constitue seulement un cas d'utilisation de celui-ci. Les traces peuvent aussi être enregistrées lors de l'exécution d'applications normales puis être analysées dans Trace

Compass. Dans ce cas, la visualisation de la trace est d'autant plus intéressante puisque les applications exécutées seront plus complexes et risquent plus de contenir des problèmes dont la cause serait difficile à identifier sans l'outil. La partie d'analyse des résultats du banc d'essai montrant chacun des intervalles mesurés sur une ligne du temps peut être utilisée pour mesurer des intervalles quelconques dans le système d'exploitation ou les applications exécutées. Par exemple, il est possible d'insérer des points de trace à l'entrée et la sortie de certaines fonctions d'intérêt du système d'exploitation ou d'applications afin d'en mesurer le temps d'exécution.

5.1.2 Banc d'essai

Dans la revue de littérature, plusieurs bancs d'essai pour les systèmes temps réel ont été présentés. Aucun d'entre eux n'est cependant adapté pour les systèmes avec partitionnement spatial et temporel. De plus, les méthodes de mesure des bancs d'essai existants ne sont pas assez précises pour caractériser fidèlement les performances des systèmes étudiés. Dans notre solution, nous proposons une approche qui permet d'enregistrer toutes les occurrences des métriques étudiées de façon à produire un rapport statistique détaillé des résultats. Ainsi, ceux-ci permettent des analyses plus complètes sur le déterminisme des systèmes ainsi que sur leur pire cas de temps d'exécution observé. Notre proposition de banc d'essai se distingue aussi des autres par son intégration optionnelle avec un système d'analyse de traces. Grâce à l'outil développé, nous pouvons rapidement identifier les scénarios d'exécution problématiques créés par l'exécution du banc d'essai.

Dans la seconde partie de l'article, nous avons présenté les résultats obtenus grâce au banc d'essai pour deux types de systèmes d'exploitation temps réel. Dans la première expérience, nous avons comparé les performances de FreeRTOS, RTEMS et Linux. Une conclusion particulièrement intéressante de ces résultats est la variation du déterminisme du coût des services testés entre Linux et les deux autres systèmes d'exploitation. En effet, comme FreeRTOS et RTEMS sont conçus depuis leurs débuts pour gérer des applications temps réel, la déviation standard des résultats est beaucoup plus petite pour ceux-ci, même pour les services ayant une performance moyenne similaire, comme l'incrémentation ou la décrémentation de sémaphores. Les résultats des coûts moyens obtenus sont cohérents avec la complexité des systèmes étudiés. En général, FreeRTOS, le plus simple des trois, impose le coût le plus faible pour l'utilisation des services, suivi par RTEMS et Linux. Contrairement à FreeRTOS, RTEMS est un système d'exploitation supportant les processeurs multicoeurs et Linux supporte les processeurs multicoeurs en plus de l'isolation spatiale entre le code usager et le code noyau. Dans la seconde expérience, nous avons étudié les performances de Linux et Xen

pour le partitionnement spatial et temporel. Nous avons aussi étudié l'effet de l'exécution de partitions concurrentes sur ces deux systèmes. Puisque ceux-ci n'évoluent pas dans des domaines critiques et que leur objectif premier n'est pas la robustesse du partitionnement, les interférences observées lors de l'utilisation de l'hyper-threading étaient à prévoir. En effet, à notre connaissance, Linux ou Xen n'implémente par défaut aucun mécanisme de verrou ou de partitionnement de cache, qui pourrait limiter ces effets comme nous l'avons mentionné au cours de la revue de littérature.

CHAPITRE 6 CONCLUSION

Le dernier chapitre de ce mémoire propose tout d'abord un résumé des travaux accomplis durant le projet. Nous exposons ensuite les limitations de la solution et discutons des améliorations possibles à celle-ci.

6.1 Synthèse des travaux

Au cours du projet, un environnement permettant l'analyse de traces ainsi que la caractérisation des performances de systèmes d'exploitation avec partitionnement spatial et temporel a été développé. Cet environnement est composé d'un plugiciel au logiciel à code source ouvert Trace Compass ainsi que d'un banc d'essai facilement portable. La vue que nous avons développée pour Trace Compass permet la visualisation de l'ordonnancement des partitions ainsi que des processus contenus à l'intérieur des partitions. De plus, la différence entre le temps alloué à chacune des partitions et celui utilisé peut être affichée. L'information est présentée sous forme d'une ligne du temps où l'axe des Y montre le nom des processus et des partitions et l'axe des X affiche l'état de ceux-ci à travers le temps. Lorsqu'un processeur multicoeur est utilisé, l'ordonnancement des partitions est regroupé par coeur. Le banc d'essai développé permet de mesurer les performances des services généralement offerts par les systèmes d'exploitation temps réel comme l'ordonnancement coopératif, la manipulation de sémaphores ou de mutex et la communication entre les tâches. En plus de ceci, nous avons créé un scénario de test qui permet d'observer les différences temporelles entre le plan d'exécution prévu pour les partitions et l'exécution réelle. La réalisation de ces deux éléments remplit les deux premiers objectifs proposés en introduction du mémoire.

Ensuite, nous avons accompli une analyse comparative de performances pour deux groupes de systèmes d'exploitation. Dans la première étude, nous avons étudié les performances de FreeRTOS, RTEMS et Linux sur un Raspberry Pi 2B+. Les détails des résultats sont présentés dans l'article du chapitre 4. Comme ces systèmes ne permettent pas le partitionnement spatial et temporel, les résultats sont limités à la première section du banc d'essai. La seconde expérience vérifie les performances du partitionnement temporel sur Linux et Xen. Le partitionnement temporel sur Linux est atteint à l'aide de groupes de contrôle. Pour Xen, ce type de partitionnement est possible en sélectionnant la politique d'ordonnancement ARINC 653. La configuration exacte pour obtenir les expériences est détaillée dans le chapitre 3. La réalisation de cette étude comparative remplit le troisième objectif du mémoire.

Finalement, nous avons montré comment l'outil d'analyse de traces proposé peut être utilisé afin de comprendre les résultats du banc d'essai. Notre étude démontre que la visualisation de traces permet d'identifier rapidement les sources de problèmes de performances intermittents. Par exemple, dans la dernière partie de l'article, nous montrons comment les résultats du banc d'essai sont affectés par l'exécution simultanée d'une seconde partition. Sans l'utilisation de l'outil, il aurait sans aucun doute été plus difficile d'identifier les circonstances causant l'augmentation du coût de changement de contexte. Ceci complète le dernier objectif que nous avons établi pour le projet.

6.2 Limitations de la solution proposée

Du côté de l'outil de visualisation de traces, la technique proposée est un premier pas afin d'adapter les méthodes connues pour les systèmes avec partitionnement spatial et temporel, mais reste assez simple. La vue sur une ligne du temps est efficace pour l'analyse manuelle, mais est plus difficile à naviguer pour des traces de grande taille. Il serait intéressant d'envisager des techniques d'automatisation pour le minage de patrons d'exécution afin d'identifier des cas d'exécutions problématiques. Notre solution permet ceci partiellement lorsque les applications étudiées produisent des paires d'évènements pour marquer des intervalles. Nous devons aussi souligner que l'obtention de toutes les données nécessaires pour le bon fonctionnement de l'outil demande une configuration bien précise. Pour Linux, une application doit lire le contenu du système de fichier des groupes de contrôle et pour Xen une application de synchronisation doit s'exécuter dans tous les domaines utilisés. Bien que ces applications supplémentaires ne sont pas exécutées pour obtenir les résultats du banc d'essai présenté dans l'article, il serait intéressant d'approfondir le surcoût de notre solution afin de conclure sur sa viabilité dans un environnement de production.

Le banc d'essai proposé est valable et utile afin d'estimer ou de comparer les performances de systèmes d'exploitation. Par contre, nous ne pouvons pas affirmer qu'il couvre tous les cas d'exécutions possibles dans une application réelle. Par conséquent, les résultats produits par celui-ci ne peuvent pas être considérés comme une borne supérieure du pire cas de temps d'exécution. De plus, l'introduction de la fine couche d'abstraction qui nous permet d'assurer la portabilité du banc d'essai a nécessairement un surcoût. La plupart des fonctions de la couche d'abstraction ne sont qu'une correspondance directe vers l'interface du système d'exploitation testé. Considérant ceci, il serait intéressant de considérer le remplacement de certaines fonctions par des macros ou par des fonctions *inline*. Ceci permettrait de limiter l'interférence de la couche d'abstraction avec les résultats. Finalement, le banc d'essai se concentre sur quelques services que nous avons identifiés comme étant les plus communs.

L'ajout de nouveau scénario de tests peut être fait sans trop de difficulté en suivant la structure des tests déjà définis.

6.3 Améliorations futures

Des améliorations seraient possibles pour la vue Trace Compass afin de fournir plus de statistiques sur l'exécution des applications. Par exemple, avec l'information présente dans les traces, nous pourrions calculer les temps d'exécution ou d'attente de chacun des processus ainsi que le temps véritablement utilisé par les processus dans les partitions par rapport au temps alloué. Ceci donnerait à l'utilisateur de l'information pour optimiser le plan d'exécution des partitions.

Ensuite, l'outil ne considère en ce moment que l'exécution sur les coeurs réels du processeur. L'ordonnancement des partitions est présenté par rapport à ceux-ci. Dans un système avec partitionnement spatial et temporel, l'ordonnancement des processus à l'intérieur des partitions se fait généralement sur des coeurs virtuels ou logiques. Il serait intéressant de proposer une visualisation de l'ordonnancement de ces processus sur les coeurs virtuels par partition.

Nous pourrions aussi explorer l'ajout de valeurs obtenues des compteurs de performance du processeur dans la visualisation. Par exemple, le nombre de défauts de cache observés pourrait être ajouté à chaque évènement de changement de contexte. Ensuite, l'information pourrait être affichée dans un histogramme qui montre l'évolution des fautes de caches dans le temps selon les coeurs. Cette information serait utile aux développeurs pour l'optimisation du temps d'exécution et pour identifer des problèmes d'interférence entre les partitions.

Finalement, une analyse plus poussée des méthodes de communication entre les partitions serait utile pour les développeurs. Le standard ARINC 653 définit deux types de ports pour l'échange de messages entre les partitions : les ports de queues (*queuing ports*) et les ports d'échantillonnage (*sampling ports*). Les messages dans les ports d'échantillonnage peuvent être écrasés, il serait donc intéressant pour le développeur de voir le nombre de messages manqués par la partition chargé de les recevoir. Les ports de queue peuvent contenir un nombre maximal de messages avant que l'envoi et la réception ne soient bloquants. Par conséquent, il pourrait être utile d'afficher aux développeurs le temps passé en attente par les processus pour l'envoi ou la réception de messages. Puisque les partitions sont ordonnancées de manière statique, cette information pourra être utile afin d'identifier un problème de fréquence d'ordonnancement des partitions dans le plan d'exécution choisi.

RÉFÉRENCES

[1] H. Nemati et M. R. Dagenais, "Virtual CPU state detection and execution flow analysis by host tracing," dans *2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud)*. IEEE, 2016, p. 7–14.

[2] C. Biancheri, N. E. Jivan et M. R. Dagenais, "Multilayer virtualized systems analysis with kernel tracing," dans *2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*. IEEE, 2016, p. 1–6.

[3] P. J. Prisaznuk, "ARINC 653 role in integrated modular avionics (IMA)," dans *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*. IEEE, 2008, p. 1–E.

[4] C. B. Watkins et R. Walter, "Transitioning from federated avionics architectures to integrated modular avionics," dans *2007 IEEE/AIAA 26th Digital Avionics Systems Conference*. IEEE, 2007, p. 2–A.

[5] A. Cook, "ARINC 653—challenges of the present and future," *Microprocessors and Microsystems*, vol. 19, n°. 10, p. 575–579, 1995.

[6] *AVIONICS APPLICATION SOFTWARE STANDARD*, Norme 653P0-1, 2015.

[7] J. Rufino, S. Filipe, M. Coutinho, S. Santos et J. Windsor, "ARINC 653 interface in RTEMS," dans *Proc. DASIA*, 2007.

[8] X. Jean, D. Faura, M. Gatti, L. Pautet et T. Robert, "Ensuring robust partitioning in multicore platforms for ima systems," dans *2012 IEEE/AIAA 31st Digital Avionics Systems Conference (DASC)*. IEEE, 2012, p. 7A4–1.

[9] J. Rushby, "Partitioning in avionics architectures : Requirements, mechanisms, and assurance," SRI INTERNATIONAL MENLO PARK CA COMPUTER SCIENCE LAB, Rapport technique, 2000.

[10] R. Fuchsen, "How to address certification for multi-core based IMA platforms : Current status and potential solutions," dans *29th Digital Avionics Systems Conference*. IEEE, 2010, p. 5–E.

[11] V. Suhendra et T. Mitra, "Exploring locking & partitioning for predictable shared caches on multi-cores," dans *2008 45th ACM/IEEE Design Automation Conference*. IEEE, 2008, p. 300–303.

[12] J. Delange et L. Lec. (2011) POK, an ARINC653-compliant operating system released under the BSD license. [En ligne]. Disponible : https://static.lwn.net/images/conf/rtlws-2011/proc/Delange_POK.pdf

[13] K. Mallachiev, N. Pakulin et A. V. Khoroshilov, “Design and architecture of real-time operating system,” , vol. 28, n°. 2, 2016.

[14] F. Reghenzani, G. Massari et W. Fornaciari, “The real-time linux kernel : A survey on PREEMPT_RT,” *ACM Computing Surveys (CSUR)*, vol. 52, n°. 1, p. 18, 2019.

[15] P. Bellasi, G. Massari et W. Fornaciari, “Effective runtime resource management using linux control groups with the BarbequeRTRM framework,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 14, n°. 2, p. 39, 2015.

[16] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt et A. Warfield, “Xen and the art of virtualization,” dans *ACM SIGOPS operating systems review*, vol. 37, n°. 5. ACM, 2003, p. 164–177.

[17] S. H. VanderLeest, “ARINC 653 hypervisor,” dans *29th Digital Avionics Systems Conference*. IEEE, 2010, p. 5–E.

[18] M. Masmano, I. Ripoll, A. Crespo et J. Metge, “Xtratum : a hypervisor for safety critical embedded systems,” dans *11th Real-Time Linux Workshop*. Citeseer, 2009, p. 263–272.

[19] S. Edgar et A. Burns, “Statistical analysis of WCET for scheduling,” dans *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001)(Cat. No. 01PR1420)*. IEEE, 2001, p. 215–224.

[20] R. Kirner et P. Puschner, “Discussion of misconceptions about WCET analysis,” dans *WCET*, 2003, p. 61–64.

[21] J. Abella, D. Hardy, I. Puaut, E. Quiñones et F. J. Cazorla, “On the comparison of deterministic and probabilistic WCET estimation techniques,” dans *2014 26th Euromicro Conference on Real-Time Systems*. IEEE, 2014, p. 266–275.

[22] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra *et al.*, “The worst-case execution-time problem—overview of methods and survey of tools,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, n°. 3, p. 36, 2008.

[23] F. Wartel, L. Kosmidis, C. Lo, B. Triquet, E. Quinones, J. Abella, A. Gogonel, A. Baldovin, E. Mezzetti, L. Cucu *et al.*, “Measurement-based probabilistic timing analysis : Lessons from an integrated-modular avionics case study,” dans *2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*. IEEE, 2013, p. 241–248.

[24] E. Mezzetti et T. Vardanega, *On the industrial fitness of WCET analysis*. na, 2011.

[25] S. Bünte, M. Zolda et R. Kirner, “Let’s get less optimistic in measurement-based timing analysis,” dans *2011 6th IEEE International Symposium on Industrial and Embedded Systems*. IEEE, 2011, p. 204–212.

[26] F. J. Cazorla, E. Quiñones, T. Vardanega, L. Cucu, B. Triquet, G. Bernat, E. Berger, J. Abella, F. Wartel, M. Houston *et al.*, "Proartis : Probabilistically analyzable real-time systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 12, nº. 2s, p. 94, 2013.

[27] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quiñones et F. J. Cazorla, "Measurement-based probabilistic timing analysis for multi-path programs," dans *2012 24th euromicro conference on real-time systems*. IEEE, 2012, p. 91–101.

[28] O. Kotaba, J. Nowotsch, M. Paulitsch, S. M. Petters et H. Theiling, "Multicore in real-time systems–temporal isolation challenges due to shared resources," dans *Workshop on industry-driven approaches for cost-effective certification of safety-critical, mixed-criticality systems*, 2013.

[29] S. Wegener, "Towards multicore WCET analysis," dans *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[30] M. Panić, E. Quiñones, P. G. Zavkov, C. Hernandez, J. Abella et F. J. Cazorla, "Parallel many-core avionics systems," dans *2014 International Conference on Embedded Software (EMSOFT)*. IEEE, 2014, p. 1–10.

[31] O. M. dos Santos et A. Wellings, "Run time detection of blocking time violations in real-time systems," dans *2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE, 2008, p. 347–356.

[32] A. I. Kistijantoro et A. Gilbran, "Improving ARINC 653 system reliability by using fault-tolerant partition scheduling," dans *2018 5th International Conference on Advanced Informatics : Concept Theory and Applications (ICAICTA)*. IEEE, 2018, p. 182–187.

[33] *AVIONICS APPLICATION SOFTWARE STANDARD*, Norme 653P1-4, 2015.

[34] O.-K. Ha, G. M. Tchamgoue, J.-B. Suh et Y.-K. Jun, "On-the-fly healing of race conditions in ARINC-653 flight software," dans *29th Digital Avionics Systems Conference*. IEEE, 2010, p. 5–A.

[35] A. Goldberg et G. Horvath, "Software fault protection with ARINC 653," dans *2007 IEEE Aerospace Conference*. IEEE, 2007, p. 1–11.

[36] A. Dubey, G. Karsai et N. Mahadevan, "Model-based software health management for real-time systems," dans *2011 Aerospace Conference*. IEEE, 2011, p. 1–18.

[37] J. Littlefield-Lawwill et L. Kinnan, "System considerations for robust time and space partitioning in integrated modular avionics," dans *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*. IEEE, 2008, p. 1–B.

[38] L. Kinnan, J. Wlad et P. Rogers, “Porting applications to an ARINC 653 compliant IMA platform using VxWorks as an example,” dans *The 23rd Digital Avionics Systems Conference (IEEE Cat. No. 04CH37576)*, vol. 2. IEEE, 2004, p. 10–B.

[39] B. Leiner, M. Schlager, R. Obermaisser et B. Huber, “A comparison of partitioning operating systems for integrated systems,” dans *International Conference on Computer Safety, Reliability, and Security*. Springer, 2007, p. 342–355.

[40] R. L. Alena, J. P. Ossenfort, K. I. Laws, A. Goforth et F. Figueroa, “Communications for integrated modular avionics,” dans *2007 IEEE Aerospace Conference*. IEEE, 2007, p. 1–18.

[41] V. Bos, T. Vepsäläinen, Y. Prokhorova et T. Latvala, “Time and space partitioning using on-board software reference architecture,” dans *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2016, p. 17–20.

[42] N. Badache, K. Jaffrès-Runser, J.-L. Scharbarg et C. Fraboul, “End-to-end delay analysis in an integrated modular avionics architecture,” dans *2013 IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA)*. IEEE, 2013, p. 1–4.

[43] P.-M. Fournier et M. R. Dagenais, “Analyzing blocking to debug performance problems on multi-core systems,” *ACM SIGOPS Operating Systems Review*, vol. 44, n°. 2, p. 77–87, 2010.

[44] R. Fahem, “Points de trace statiques et dynamiques en mode noyau,” Mémoire de maîtrise, École Polytechnique de Montréal, 2012.

[45] M. Gebai et M. R. Dagenais, “Survey and analysis of kernel and userspace tracers on linux : Design, implementation, and overhead,” *ACM Computing Surveys (CSUR)*, vol. 51, n°. 2, p. 26, 2018.

[46] S. Rostedt. (2008) ftrace - function tracer. [En ligne]. Disponible : https://www.kernel.org/doc/Documentation/trace/ftrace.txt

[47] ——, “Finding origins of latencies using ftrace,” dans *11th Real-Time Linux Workshop*, 2009, p. 28–30.

[48] M. Desnoyers, “Low-impact operating system tracing,” Thèse de doctorat, École Polytechnique de Montréal, 2009.

[49] M. Desnoyers et M. R. Dagenais, “Lockless multi-core high-throughput buffering scheme for kernel tracing.” *Operating Systems Review*, vol. 46, n°. 3, p. 65–81, 2012.

[50] A. C. De Melo, “The new linux’perf’tools,” dans *Slides from Linux Kongress*, vol. 18, 2010.

[51] B. Gregg. (2019) perf examples. [En ligne]. Disponible : https://percepio.com/2018/11/07/visualizing-state-machines/

[52] Efficios. barectf. [En ligne]. Disponible : https://github.com/efficios/barectf

[53] Percepio. Tracealyzer for FreeRTOS. [En ligne]. Disponible : https://percepio.com/tz/freertostrace/

[54] J. Kraft, A. Wall et H. Kienle, "Trace recording for embedded systems : Lessons learned from five industrial projects," dans *International Conference on Runtime Verification*. Springer, 2010, p. 315–329.

[55] M. Holenderski, M. Van Den Heuvel, R. J. Bril et J. J. Lukkien, "Grasp : Tracing, visualizing and measuring the behavior of real-time systems," dans *International workshop on analysis tools and methodologies for embedded and real-time systems (WATERS)*, 2010, p. 37–42.

[56] M.-s. Kang, M.-H. Kang, O.-K. Ha et Y.-K. Jun, "Conpathview : A visualization tool for debugging race conditions in event synchronization of ARINC 653 applications," *International Journal of Software Engineering and Its Applications*, vol. 10, n°. 2, p. 65–76, 2016.

[57] G. M. Tchamgoue, L. Gan, O.-K. Ha, S.-W. Yang et Y.-K. Jun, "Visualizing concurrency faults in arinc-653 real-time applications," dans *2012 IEEE/AIAA 31st Digital Avionics Systems Conference (DASC)*. IEEE, 2012, p. 9E6–1.

[58] G. M. Tchamgoue, O.-K. Ha, K.-H. Kim et Y.-K. Jun, "A framework for on-the-fly race healing in ARINC-653 applications," *International Journal of Hybrid Information Technology*, vol. 4, n°. 2, p. 1–12, 2011.

[59] A. Sailer, M. Deubzer, G. Lüttgen et J. Mottok, "Coretana : A trace analyzer for reverse engineering real-time software," dans *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, p. 657–660.

[60] Percepio. How to visualize response times in FreeRTOS. [En ligne]. Disponible : https://percepio.com/percepio-tracealyzer-whitepaper.pdf

[61] ——. (2018) Analyzing state machines. [En ligne]. Disponible : https://percepio.com/2018/11/07/visualizing-state-machines/

[62] Eclipse. Trace compass. [En ligne]. Disponible : https://www.eclipse.org/tracecompass/

[63] A. Montplaisir-Gonçalves, N. Ezzati-Jivan, F. Wininger et M. R. Dagenais, "State history tree : an incremental disk-based data structure for very large interval data," dans *2013 International Conference on Social Computing*. IEEE, 2013, p. 716–724.

[64] F. Giraldeau et M. Dagenais, “Wait analysis of distributed systems using kernel tracing,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, n°. 8, p. 2450–2461, 2015.

[65] K. Kouame, N. Ezzati-Jivan et M. R. Dagenais, “A flexible data-driven approach for execution trace filtering,” dans *2015 IEEE International Congress on Big Data*. IEEE, 2015, p. 698–703.

[66] M. Côté et M. R. Dagenais, “Problem detection in real-time systems by trace analysis,” *Advances in Computer Engineering*, vol. 2016, 2016.

[67] Q. S. Systems. Analyzing your system with kernel tracing. [En ligne]. Disponible : http://www.qnx.com/developers/docs/6.3.0SP3/ide_en/user_guide/sysprof.html

[68] R. System. Rapitask. [En ligne]. Disponible : https://www.rapitasystems.com/products/rapitask

[69] W. River. Wind river workbench 3.3. [En ligne]. Disponible : https://www.windriver.com/products/product-notes/PN_Workbench_0611/PN_Workbench_0611.pdf

[70] K. Skadron, M. Martonosi, D. I. August, M. D. Hill, D. J. Lilja et V. S. Pai, “Challenges in computer architecture evaluation,” *Computer*, vol. 36, n°. 8, p. 30–36, 2003.

[71] R. P. Weicker, “An overview of common benchmarks,” *Computer*, vol. 23, n°. 12, p. 65–75, 1990.

[72] A. Garcia-Martinez, J. F. Conde et A. Vina, “A comprehensive approach in performance evaluation for modern real-time operating systems,” dans *Proceedings of EUROMICRO 96. 22nd Euromicro Conference. Beyond 2000 : Hardware and Software Design Strategies*, Sep. 1996, p. 61–68.

[73] R. P. Kar, “Implementing the rhealstone real-time benchmark,” *Dr. Dobb’s Journal*, vol. 15, n°. 4, p. 46–55, 1990.

[74] T. N. B. Anh et S.-L. Tan, “Real-time operating systems for small microcontrollers,” *IEEE micro*, vol. 29, n°. 5, p. 30–45, 2009.

[75] D. P. B. Renaux, “Comparative performance evaluation of CMSIS-RTOS,” dans *2014 Brazilian Symposium on Computing Systems Engineering*. IEEE, 2014, p. 126–131.

[76] F. Nicodemos, O. Saotome et G. Lima, “RTEMS core analysis for space applications,” dans *2013 III Brazilian Symposium on Computing Systems Engineering*. IEEE, 2013, p. 125–130.

[77] G. Kasten, D. Howard et B. Walsh, “Rhealstone recommendations,” *Dr. Dobb’s Journal*, vol. 15, n°. 9, p. 8–12, 1990.

[78] E. McRae, “Benchmarking real-time operating systems,” *Dr Dobb’s Journal-Software Tools for the Professional Programmer*, vol. 21, n°. 5, p. 48–59, 1996.

[79] N. Weiderman, “Hartstone : synthetic benchmark requirements for hard real-time applications,” Software Engineering Institute, Carnegie Mellon University, Rapport technique, 1990.

[80] N. I. Kamenoff et N. H. Weiderman, “Hartstone distributed benchmark : requirements and definitions,” dans *[1991] Proceedings Twelfth Real-Time Systems Symposium*. IEEE, 1991, p. 199–208.

[81] C. Tres, L. B. Becker et E. Nett, “Real-time tasks scheduling with value control to predict timing faults during overload,” dans *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC’07)*. IEEE, 2007, p. 354–358.

[82] The Linux Foundation. (2019) Rt-tests. [En ligne]. Disponible : https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/rt-tests

[83] R. Beamonte, F. Giraldeau et M. Dagenais, “High performance tracing tools for multicore linux hard real-time systems,” dans *Proceedings of the 14th Real-Time Linux Workshop*. OSADL, 2012.

[84] Lamie, William and Carbone, John. (2007) Measure your rtos’s real-time performance. [En ligne]. Disponible : https://www.embedded.com/design/operating-systems/4007081/Measure-your-RTOS-s-real-time-performance

[85] T. Gomes, F. Salgado, A. Tavares et J. Cabral, “CUTE mote, a customizable and trustable end-device for the internet of things,” *IEEE Sensors Journal*, vol. 17, n°. 20, p. 6816–6824, 2017.

[86] E. H. Sibley, “Dhrystone : A synthetic systems,” *Communications of the ACM*, vol. 27, n°. 10, 1984.

[87] S. Han et H.-W. Jin, “Resource partitioning for integrated modular avionics : comparative study of implementation alternatives,” *Software : Practice and Experience*, vol. 44, n°. 12, p. 1441–1466, 2014.

[88] J. Desfossez, M. Desnoyers et M. R. Dagenais, “Runtime latency detection and analysis,” *Software : Practice and Experience*, vol. 46, n°. 10, p. 1397–1409, 2016.

[89] F. Rajotte et M. R. Dagenais, “Real-time linux analysis using low-impact tracer,” *Advances in Computer Engineering*, vol. 2014, 2014.

[90] L. Sha, R. Rajkumar et J. P. Lehoczky, “Priority inheritance protocols : An approach to real-time synchronization,” *IEEE Transactions on computers*, vol. 39, n°. 9, p. 1175–1185, 1990.

[91] G. van Loo. (2014) Quad-A7 control. [En ligne]. Disponible : https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2836/QA7_rev3.4.pdf

[92] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge et R. B. Brown, "MiBench : A free, commercially representative embedded benchmark suite," dans *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*. IEEE, 2001, p. 3–14.